Esa Vanhanen-Varho

REGRESSIOTESTAUKSEN TEHOSTAMINEN PALVELURIIPPUVAISESSA YMPÄRISTÖSSÄ – TAPAUS MERIMIESELÄKEKASSA

Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan tutkinto-ohjelma

Diplomityö jätetty tarkastettavaksi opinnäytteenä diplomi-insinöörin tutkintoa varten Espoossa 30.4.2010.

Ohjaaja: DI Antti Sulanto

Valvoja: Prof. Tuomas Aura

Aalto University
School of Science
and Technology

AALTO-YLIOPISTO
TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Esa Vanhanen-Varho

Työn nimi: Regressiotestauksen tehostaminen palveluriippuvaisessa ympäristössä – tapaus Merimieseläkekassa

Päivämäärä: 30.4.2010 Kieli: suomi Sivumäärä: 4 + 91

Informaatio- ja luonnontieteiden tiedekunta

Tietotekniikan tutkinto-ohjelma

Tietoliikenneohjelmistot

Valvoja: Prof. Tuomas Aura

Ohjaaja: DI Antti Sulanto

Tässä diplomityössä tarkastellaan regressiotestauksen tehostamista työeläkelaitoksen (Merimieseläkekassa) järjestelmäuudistusprojektin yhteydessä. Erikoispiirteenä uudistettavassa järjestelmässä ovat sen lukuisat toiminnallisuuden kannalta kriittiset liittymät sekä eläkealan yhteisiin järjestelmiin että sisäisiin sovelluksiin www-sovelluspalveluiden (web services) kautta. Regressiotestauksen tehostamistarve ennen järjestelmän käyttöönottoa on projektin ylläpitovaiheen kannalta keskeinen tekijä, jotta järjestelmän jatkokehityksen yhteydessä ohjelmiston testaus voidaan suorittaa kehitysvaihetta selvästi pienemmillä resursseilla. Työn tarkoituksena oli löytää tehostamisen mahdollistavia menetelmiä sekä tarvittaessa valita tai suunnitella niiden käyttöönoton vaatimia työkaluja.

Työn alkuosa sisältää laajahkon kirjallisuuskatsauksen testauksen ja testiautomaation teoriaan. Työn loppuosassa tätä tietoa on sovellettu käynnissä olevaan kehitysprojektiin laatimalla suunnitelma kehitysprosessiin tehtävistä muutoksista sekä tehostamistoimiin tarvittavista työkaluista.

Www-sovellusrajapintojen sekä .NET:in WCF-rajapintojen yleisyydestä johtuen keskeiseksi testausmalliksi muodostui palvelurajapintojen kautta siirrettäviä XML-sanomia tukevan työkalun laatiminen. Tälle työkalulle löytyi pohjaratkaisu avoimen lähdekoodin BizUnit-projektista, mikä mahdollistaa myös testien automatisoinnin sekä palvelupyyntöjen ketjuttamisen monivaiheiseksi prosessiksi. Työn aikana tunnistettiin ja suunniteltiin laajennukset, jotka tukevat entistä paremmin kohdeympäristön testaamista BizUnitin avulla. Lisäksi diplomityössä syntyi XML-skeemojen muutosvaikutuksia analysoiva prosessi, joka helpottaa sisäisten järjestelmien muutostarpeen arviointia ulkoisten rajapintojen muuttuessa.

Työn käytännön soveltuvuus konkretisoituu vasta käyttöönottovaiheen jälkeen vuoden 2010-2011 vaihteessa. Tätä ennen suunnitelmia viedään eteenpäin pienien pilottihankkeiden muodossa aiemmin tuotantokäyttöön otetuille järjestelmille.

Avainsanat: ohjelmistotestaus, regressiotestaus, testiautomaatio, .NET, www-sovelluspalvelut, testaus, automatisointi

AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS
ABSTRACT

Author: Esa Vanhanen-Varho

Thesis Title: Improving regression testing in a service dependent environment – case Seafarer's Pension Fund

Date: 2010-04-30 Language: Finnish Number of Pages: 4 + 91

Faculty of Information and Natural Sciences

Department of Computer Science and Engineering

Data Communications Software

Supervisor: Prof. Tuomas Aura

Instructor: Antti Sulanto, M.Sc. (Tech).

The main objective of this thesis was to study how regression testing could be improved before maintenance phase during Seafarer's Pension Fund system renewal project by identifying and planning possible process improvements and tools that make testing more efficient. The renewed systems are heavily dependent on both external and internal interfaces based on web services. More efficient regression testing before maintenance phase is vital because the size of the test team will be significantly lower than during the ongoing implementation phase.

This theses starts with a relatively wide literature study on testing and test automation. In the final part this theory has been applied for the chosen project environment by forming a plan on required changes to development and testing process and required tools.

A general tool for testing both external and internal web service and WCF interfaces with XML-based messages was identified as a required improvement as these kinds of interfaces are very common both externally and internally. The solution was planned on top of open source tool BizUnit, which enables also test automation and chaining of the service calls to create a multi-phased process. During the work required extensions were also identified and planned. Another planned tool will help to efficiently analyze the impact of changes in the service schemas for internal applications.

Evaluation of the planned changes takes place in the future after the implementation project during 2011. Before that planned changes are implemented as small pilot projects for systems that have already been deployed to production environment.

Keywords: software testing, regression testing, test automation, .NET, web services, testing, automation

Alkulause

Tämä diplomityö on kirjoitettu Merimieseläkekassan järjestelmäuudistusprojektin yhteydessä työskennellessäni Avanade Finland Oy:ssä syksyn 2009 ja kevään 2010 välisenä aikana.

Haluan kiittää valvojaani professori Tuomas Auraa sekä ohjaajaani DI Antti Sulantoa kommenteista ja kannustuksesta työtä tehdessäni sekä Merimieseläkekassan Jorma Niemistä mahdollisuudesta laatia työ käynnissä olevan projektin yhteydessä.

Lisäksi haluan kiittää koko Merimieseläkekassan projektitiimiä työn aikaisista kommenteista sekä mahdollisuudesta joustaviin työaikajärjestelyihin, joita ilman opintojen loppuun saattaminen ei olisi onnistunut. Tätä helpotti myös DI 2010-projekti ja diplomityöseminaari pienryhmätapaamisineen, joihin etenkin Taina Hyppölä ja Sanna Suoranta käyttivät aikaansa.

Kiitokset myös kaikille juoksijoille, jotka tartuttivat minuun ultrakipinän. En usko, että olisin jaksanut tätä projektia loppuun asti tuntematta niitä voimavaroja, joita ihmismieli kykenee kontrolloimaan väsyneenäkin.

Suurimmat kiitokset osoitan kuitenkin äidilleni, siskolleni, appivanhemmilleni ja kaikille ystävilleni, jotka tuitte ja autoitte pitämään lapsiperhettämme pystyssä arjen askareissa – sekä tietenkin rakkaalle vaimolleni Viljalle. En ymmärrä, kuinka olet jaksanut univajeen keskellä koko pitkän vuoden. Ja pienet päivänsäteeni, Visa ja Kaisla – nyt isi ei enää ole koko ajan töissä…

Espoossa, 30.4.2010

Esa Vanhanen-Varho

Käytetyt lyhenteet ja määritelmät

.NET	.NET Framework, Microsoftin komponenttikirjasto ja ohjelmointialusta
ACA.NET	Avanade Connected Architectures®, sovelluskehitys/mallinnustyökalu
ASP.NET	Active Server Pages .NET, verkkopalvelujen ohjelmointimenetelmä .NET-ympäristöön
C#	.NET ympäristössä toimiva ohjelmointikieli
CIL	Common Intermediate Language, matalan tason välikieli, johon .NET-ohjelmien lähdekoodi käännetään ennen tavukoodin muodostamista
COTS	Commercial Off-The-Shelf, yleensä kaupallinen ohjelmistokomponentti tai sovellus, jota käytetään toteutuksen osana
CRM	Customer Relationship Management, asiakkuudehallinta(sovellus)
DTD	Document Type Definition, rakenteellisten dokumenttien syntaksin kuvauskieli
Häiriö	Ohjelman kyvyttömyys toimia vaatimusmäärittelynsä mukaisesti. Kaikki virheet ja viat eivät aina johda häiriöön. (*engl. failure*)
GUI	Graafinen käyttöliittymä (*engl. graphical user interface*)
JavaScript	Selaimessa suoritettava komentosarjakieli
MEK	Merimieseläkekassa
MSIL	Microsoft Intermediate Language, ks. CIL
REST	Representational State Transfer
SOAP	Www-sovelluspalveluiden käyttämä tiedonvaihtoprotokolla, alun perin lyhenne sanoista Simple Object Access Protocol
SSRS	SQL Server Reporting Services, SQL Serverin raportointikomponentti
Testi	Toisiinsa liittyvien testitapausten ja toimenpiteiden joukko, jonka avulla määrätty asia voidaan testata. (*engl. test*)
Testikehys	Laitteisto- ja ohjelmistoympäristö, jotka ohjelmiston tai ohjelmakomponentin testaamiseksi tarvitaan..
Testioraakkeli	Dokumentti tai ohjelma, jonka avulla testaaja voi päätellä testin onnistumisen tai epäonnistumisen. (*engl. test oracle*)
Testitapaus	Syötteiden, suoritusten aikaisten ehtojen ja odotettujen tulosten joukko, jolla on tietty tarkoitus – esimerkiksi ohjelmalle määritellyn vaatimuksen toteutumisen varmistaminen tai tietyn polun suorittaminen ohjelmakoodissa. Testitapaus tarkoittaa myös dokumenttia, jossa samat asiat kuvataan liittyen tiettyyn testattavaan komponenttiin. (*engl. test case*)

UDDI	Universal Description Discovery and Integration. OASISin ylläpitämä palvelurekisteristandardi, joka kuuluu www-sovelluspalveluiden yhteentoimivuutta edistävään WS-I suositukseen
UML	Unified Modeling Language™, Object Management Group™:n laatima mallinnuskieli, joka koostuu 14 kaaviotyypistä.
Virhe	Ihmisen tekemä erehdys tai väärinkäsitys. (*engl. error*)
Vika	Virheen vuoksi ohjelmaan syntynyt poikkeama, joka voi saada ohjelman toimimaan virheellisesti ja määritysten vastaisesti. Vika voi esiintyä myös ohjelmakoodin ulkopuolella, esimerkiksi määrittelydokumentissa. (*engl. fault, defect, bug*)
WCF	Windows Communication Foundation
WF	Windows Workflow Foundation
WSDL	Web Service Definition Language, XML-pohjainen www-sovelluspalvelujen kuvauskieli
Www-sovelluspalvelu	Www-sovelluspalvelu on ohjelmistojärjestelmä, joka on suunniteltu tukemaan koneiden välistä vuorovaikutusta tietoverkoissa yhteentoimivien WSDL-kuvauskielellä kuvattujen rajapintojen avulla. Palveluja käytetään kuvauksen mukaisilla SOAP-viesteillä, joissa yleensä siirretään XML-muotoista tietoa HTTP-protokollaa käyttäen. (W3C 2004a, *engl. web service*)
XPath	XML Path Language, XML-sanomien kyselykieli

Sisältö

1 Johdanto

1.1 Työn taustaa

Ohjelmistojen elinkaaren aikaisista kustannuksista jopa yli puolet syntyy Sommervillen (2007, s. 493-494) mukaan ohjelmiston ylläpitovaiheessa. Hän jatkaa, että suurin osa tästä työstä liittyy ohjelmiston toiminnallisuuksiin tarvittaviin muutoksiin tai lisäyksiin, eikä suinkaan ohjelmistovirheiden korjaamiseen. Ylläpitovaiheen kustannuksista jopa puolet voi liittyä ohjelmiston regressio- eli uudelleentestaukseen (Harrold 2009). Regressiotestauksella pyritään varmistamaan, että järjestelmän aiemmat toiminnallisuudet pysyvät ehjinä tehtyjen muutosten jälkeen – itse asiassa järjestelmä tulisi testata uudelleen jokaisen koodiin tai ympäristöön tehdyn muutoksen jälkeen (Kamer et al. 1999). Ylläpito- ja regressiotestaukseen liittyvien toistuvien tehtävien tehostaminen onkin järkevää kertautuvien kustannusten pienentämiseksi.

Merimieseläkekassa (MEK) huolehtii merenkulkijoiden lakisääteisestä työeläketurvasta. MEK on uudistamassa järjestelmiään aiemmista keskuskonepohjaisista ratkaisuista Microsoft-arkkitehtuuriin ja valmisohjelmistoihin perustuvaksi. Uudistustyötä on tehty vaiheittain alkaen verkkopalveluista ja yksinkertaisemmista sisäisistä sovelluksista. Parhaillaan on käynnissä toiminnallisesti merkittävimpien eläkkeisiin liittyvien hakemus- ja hoitoprosessien uudistaminen. Toimintaympäristöä on kuvattu tarkemmin luvussa 5.

Työeläkelaitokset ovat riippuvaisia keskitetyistä järjestelmistä, joissa säilytetään muun muassa henkilöiden työ- ja ansiohistoriaan liittyviä tietoja, eivätkä MEKin järjestelmät tee tästä poikkeusta. Esimerkiksi eläkehakemusprosessissa tieto hakemuksesta ja siihen liittyvistä päätöksistä päivitetään näihin keskitettyihin järjestelmiin, hakijan työhistoria noudetaan ansaintarekisteristä ja eläkkeen laskenta suoritetaan pääsääntöisesti yhteisen laskentapalvelun avulla. Yhteisten järjestelmien käyttö on järkevää, jotta pienen toimijan, kuten MEK, ei tarvitse itse toteuttaa kaikkia toimintoja omiin järjestelmiinsä itse ylläpidettäviksi.

Yhteisten järjestelmien käyttö synnyttää kuitenkin riippuvuuden näihin ulkoisiin palveluihin ja niissä tapahtuviin melko säännöllisiin muutoksiin, mikä aiheuttaa haasteita ylläpidolle. Tätä korostetaan myös tässä diplomityössä. Kaikki ulkoisten järjestelmien päivitykset eivät vaadi MEKin järjestelmien muuttamista, mutta näistä ympäristön muutoksista syntyy aina tarve järjestelmien uudelleentestaukseen, kuten yllä todettiin. Ulkoiset palvelut toimivat eräänlaisina valmiskomponentteina, joita käytetään tietämättä niiden toteutuksen yksityiskohtia, eikä niistä ei ole mahdollista saada tarkkaa tietoa suoritettujen testien osalta. Palveluiden testaaminen on myös prosessiluonteista: toimintojen testaaminen ei ole mahdollista, ellei testin kohteen alkutila ole oikea. Esimerkiksi hakemuksen peruuttamista ei voida testata, ellei sitä ole ensin luotu. Tämän vuoksi suoritettavat testit ovat varsin laajoja kokonaisuuksia.

Riippuvuus ulkoisista järjestelmistä aiheuttaa haasteita myös kehitysympäristölle. Ulkoisissa järjestelmissä esiintyvät katkokset havaittiin kehitystyön aikana varsin yleisiksi. Joskus katkokset olivat pitkiä, mikä sotki kehitys- ja testausaikataulua. Sama tilanne jatkuu oletettavasti myös ylläpitovaiheessa johtuen näiden ulkoisten järjestelmien jatkuvasta uudistamisesta. Testaukseen aiheutuvien häiriöiden ehkäisyllä olisi positiivinen vaikutus työn tehokkuuteen varsinkin erillisen testitiimin työn kannalta. Tämä edellyttäisi kuitenkin varsin laajaa järjestelmää monimutkaisten ulkoisten palveluiden jäljittelemiseksi.

Hakemusjärjestelmän kehitystyön aikana nousi esiin myös tarve osallistua suoraan ulkoisten palveluiden hyväksymistestausprosessiin, jossa testin kohteena olivat tietyt hakemuksen käsittelyyn liittyvät www-sovelluspalvelut (engl. web services). Tämä toteutettiin väliaikaisilla testiohjelmilla, joilla voitiin testata sanomaliikennettä käsin tehdyin testisanomin sekä suoraan ulkoisiin palveluihin että MEKin oman integraatiopalvelun tarjoamien www-sovelluspalvelurajapintojen kautta. Nämä hyväksymistestausprosessit toistuvat säännöllisesti ylläpitovaiheessa, ja MEKin ylläpidosta vastaava toimija osallistuu silloinkin testaukseen. Järjestelmien laajentuessa testaukseen liittyvä työmäärä kasvaa entisestään, joten hyväksymistestauksen tehostaminen on tärkeä tehtävä.

Asiaa selvitettäessä huomattiin, että www-sovelluspalvelurajapintoja käyttävää testausohjelmaa voitaisiin hyödyntää laajemminkin MEKin sisäisten järjestelmien testaamiseen, koska keskeiset valmisohjelmistot ja osa räätälöidyistä liiketoimintalogiikan toiminnallisuuksista oli jo julkaistu palveluina, ja uuden palvelun julkaiseminen jopa pelkkää testausta varten huomattiin helpoksi toimenpiteeksi.

Testausohjelmiston tarpeen lisäksi tunnistettiin uudelleentestauksessa tarvittavien testien suureen määrään, käyttöliittymätestien monimutkaisuuteen sekä ulkoisten liittymien muutosvaikutuksen arvioimisen työläyteen liittyvät haasteet. Tässä työssä pyrittiin selvittämään, kuinka näitä ongelmia olisi mahdollista ratkoa joko työkalujen avulla tai toimintatapoja muuttamalla.

1.2 Työn tavoitteet

Työn tavoitteena oli suunnitella ratkaisumalleja Merimieseläkekassan järjestelmäuudistuksen jälkeisen ylläpitovaiheen tehostamiseen regressiotestauksen näkökulmasta huomioiden johdannossa mainitut haasteet sekä ratkaisujen kustannustehokkuus.

Keskeisimmiksi ongelmiksi ympäristössä oli projektin kuluessa tunnistettu:

- Kuinka palvelupyyntöjen ja niistä muodostuvien kokonaisprosessien testausta voidaan tehostaa? Minkälaisia testaustyökaluja tämä edellyttää?
- Kuinka muutosten testaamiseen tarvittavien testien määrää voidaan rajoittaa?
- Kuinka palvelurajapintojen muutosten vaikutuksien arviointia MEKin omiin järjestelmiin voidaan helpottaa ja välttää turhia muutostöitä?
- Kuinka ulkoisten rajapintojen katkoksista aiheutuvat häiriöt kehitys- ja testausympäristöille voidaan välttää?

Ratkaisumalleja lähdettiin hakemaan laajan kirjallisuuskatsauksen kautta etsimällä regressiotestaukseen ja testauksen automatisointiin liittyviä menetelmiä, joiden soveltuvuutta MEK-projektiin arvioitiin tapauskohtaisesti. Projektin aikana käyttöön otetun ketterän kehitystavan tukemiseksi pyrittiin löytämään ratkaisuja, joiden hyödyntäminen olisi mahdollista edes osittain jo toteutusvaiheen aikana. Ratkaisujen nopeampi hyödyntäminen parantaisi samalla niiden kustannustehokkuutta. Yleisemmällä tasolla työn tavoitteena oli hakea ratkaisuja palvelurajapintoihin ja niiden muutoksiin liittyvän testauksen ongelmiin.

Keskeisenä teemana pyrittiin pitämään palvelurajapintoihin ja niiden muutoksiin liittyvän testaamisen ongelmat. Ratkaisumallien lisäksi pyrittiin suunnittelemaan ratkaisujen toteuttamiseen liittyvien työkalujen toiminnallisuudet ja arvioimaan mahdollisten valmiiden ohjelmistojen käyttökelpoisuutta toteutuksien pohjana.

1.3 Työn rajaukset

Projektin aikataulusta johtuen suunniteltujen ratkaisujen toteuttaminen ja niiden toimivuuden arviointi jäi työn ulkopuolelle. Käytännön toteutuksia tullaan suorittamaan kevään 2010 jälkeen ennen seuraavaa suurta käyttöönottovaihetta vuoden 2011 vaihteessa.

Työssä jätettiin tietoisesti huomioimatta myös monia testauksen osa-alueita, kuten esimerkiksi suorituskyky- ja turvallisuustestaus. Merimieseläkekassan käyttämistä ulkoisista palveluista työn ulkopuolelle rajattiin sellaiset verkkopalvelut tai tietokannat, joita uudistettavat järjestelmät eivät suoraan käytä.

Työkalu- ja testimenetelmiä arvioidessa prosessikuvauskieliä käyttävät menetelmät (esim. WS-BPEL) rajautuivat työn ulkopuolelle, koska näitä kuvauskieliä ei käytetä työn kohderatkaisuissa.

1.4 Työn rakenne

Työn ensimmäinen osa käsittää kirjallisuuskatsauksen, jossa perehdytään ohjelmistotestauksen periaatteisiin ensin yleisellä tasolla, minkä jälkeen aihetta laajennetaan regressiotestaukseen, automatisointiin ja palvelurajapintojen testaamiseen liittyvillä näkökulmilla. Toisena osana esitellään työn kohdeympäristö ja nykytila. Kolmantena kokonaisuutena ovat ehdotetut ratkaisut perusteluineen. Lopuksi arvioidaan ratkaisuja ja itse työn suoritusta sekä tehdään yhteenveto koko työstä jatkokehityssuunnitelmineen.

2 Ohjelmistojen testaus

Tässä luvussa käydään läpi ohjelmistojen testausta: miksi testataan, mitä ovat staattinen ja rakenteellinen testaus. Luvussa käydään myös lyhyesti läpi ohjelmistokehitysprosessin ja erilaisten testaustasojen välisen riippuvuudet. Lopuksi käsitellään regressio- eli uudelleentestausta.

2.1 Yleistä

SWEBOK (2004, luku 5) määrittelee testauksen seuraavasti: "Testaus on tehtävä, jonka avulla pyritään arvioimaan ja parantamaan ohjelmiston laatua tunnistamalla siitä vikoja ja ongelmia." Testaus on siis osa ohjelmistojen laadun varmistamista, tarkemmin sanoen se kuuluu verifioinnin ja validoinnin (V & V) tehtäviin. Suomeksi voidaan käyttää myös termejä todentaminen ja kelpuuttaminen (TTL 2007). IEEE:n standardin (Std 1059-1993) mukaan "Verifiointi ja validointi on toimintamalli, jolla ohjelmistoa voidaan arvioida sen elinkaaren ajan ja jolla pyritään varmistamaan ohjelmiston vaatimusten täyttäminen rakentamalla ohjelmistoa laadukkaasti". Verifioinnilla varmistetaan, että projektin jokaisen vaiheen tuotteet noudattavat niille aiemmin asetettuja vaatimuksia ja että ne on tuotettu noudatettavien standardien ja ohjeiden mukaan. Validoinnilla pyritään varmistamaan, että lopputuote on virallisten määritysten täyttymisen lisäksi toimiva siinä tarkoituksessa, mihin se on rakennettu. Boehm (1984) havainnollistaa eron seuraavasti: verifioinnilla varmistetaan, rakennetaanko tuote oikein, ja validoinnilla, rakennetaanko oikeaa tuotetta.

Testaus ei rajoitu pelkkään ohjelmakoodiin vaan myös muut ohjelmistoprojektissa tuotetut materiaalit, esimerkiksi määritysdokumentit, voidaan testata. Testausta ei voi pitää erillisenä projektin vaiheena vaan omana tehtävänään, joka jatkuu koko ohjelmiston elinkaaren ajan. Näihin tehtäviin kuuluu testien suorittamisen ja virheiden raportoinnin lisäksi testien suunnittelu, dokumentointi, mittaaminen ja raportointi (SWEBOK 2004, luku 5).

Myers et al.[1] (2004, s. 5-8 & 14-17) pitävät testausta pitkälti psykologisena tehtävänä. Testauksen ja testaajan tehtäväksi kuvataan usein ohjelman virheettömyyden varmistaminen, mutta täydellinen testaaminen on käytännössä mahdoton tehtävä (Kaner et al. 1999, s. 17-25). Ihminen luovuttaa helposti mahdottoman tehtävän edessä, minkä vuoksi Myers et al. (2004) kääntävät asian päinvastaiseksi: testauksen tehtävä on osoittaa, että testattava kohde *ei* toteuta sille asetettuja vaatimuksia tai tavoitteita. Näin määriteltynä onnistunut testi on sellainen, joka löytää vian. Samalla testaajan työ muuttuu mahdolliseksi ja motivoivammaksi, koska työllä on selvä tavoite: löytää vikoja ja huolehtia siitä, että ne tulevat korjatuksi (Kaner et al. 1999, s. 25). Psykologisista syistä Myers et al. (2004, s. 16-17) suosittelevat myös, että ohjelmoijan itsensä ei pitäisi toimia oman koodinsa ainoana testaajana, sillä ihminen ei huomaa helposti omia virheitään. Toisen henkilön suorittama testaus on tästä syystä tehokkaampaa. He vievät ajatuksen jopa niin pitkälle, että testaajien pitäisi tulla eri organisaatiosta kuin toteuttajien. Muutoin vaarana on, että organisaation tulostavoitteet menevät testauksen tavoitteiden edelle. Näistä ristiriidoista huolimatta testaaminen ohjelmoijan tai organisaation omasta toimesta on mahdollista, muttei välttämättä yhtä tehokasta kuin puolueettoman tahon suorittamana.

[1] Suurin osa viitteistä Myers et al. (2004) löytyy sellaisenaan myö alkuperäisestä Myers (1979):stä

Testaaminen vaatii toiminnan suunnittelemista ja raportointia. Testausdokumenteille on olemassa IEEE:n standardi (Std 829-2008). Kaner et al. (1999, s. 247) pitää standardia hyvänä pohjana, mutta huomauttaa, että sitä kannattaa soveltaa omiin tarpeisiin karsimalla oman ympäristön kannalta turhia yksityiskohtaisuuksia pois, koska edes standardi ei vaadi kaikkien sisältämiensä dokumenttien tuottamista.

Ohjelmistotestauksen perustermeillä on varsinkin englanniksi osin päällekkäisiä merkityksiä eri lähteissä. Tässä työssä käytetyt suomennetut (TTL 2007) perustermit ovat:

- *Virhe* (engl. error) on ihmisen tekemä erehdys tai väärinkäsitys (Burnstein 2003, s. 20; IEEE Std 610.12-1990).
- *Vika, defekti* (engl. fault, defect, bug) on virheen vuoksi ohjelmistoon syntynyt poikkeama, joka voi saada ohjelman toimimaan virheellisesti ja määritysten vastaisesti. Vika voi esiintyä myös ohjelmakoodin ulkopuolella, esimerkiksi määrittelydokumentissa (Burnstein 2003, s. 20).
- *Häiriö* (engl. failure) on ohjelmiston kyvyttömyys toimia vaatimusmäärittelynsä mukaisesti (IEEE Std 610.12-1990). Kaikki virheet ja viat eivät aina johda häiriöön (Burnstein 2003, s. 20).
- *Testitapaus* (engl. test case) on syötteiden, suoritusten aikaisten ehtojen ja odotettujen tulosten joukko, jolla on tietty tarkoitus - esimerkiksi ohjelmalle määritellyn vaatimuksen toteutumisen varmistaminen tai tietyn polun suorittaminen ohjelmakoodissa. Testitapaus tarkoittaa myös dokumenttia, jossa samat asiat kuvataan liittyen tiettyyn testattavaan komponenttiin. (IEEE Std 610.12-1990)

Testauksen taloudellinen ongelma voidaan tiivistää kysymykseen, miten testaaminen voidaan suorittaa rajallisen budjetin puitteissa mahdollisimman tehokkaasti ja siten löytää suurin mahdollinen määrä virheitä rajallisella määrällä testejä (Myers et al. 2004, s. 10-11). Mitä aikaisemmin testausta suoritetaan, sitä aiemmin on mahdollista löytää ohjelmistosta vikoja. Vaikka testaamisesta aiheutuu kustannuksia, aikaisempi vikojen löytäminen tuo lopulta säästöä, sillä Boehm ja Basili (2001) ovat vahvistaneet havainnot, että vian korjaaminen ohjelman julkaisun jälkeen voi olla jopa 100 kertaa kalliimpaa kuin sen aiheuttaneen määrittely- tai suunnitteluvian korjaaminen.

Testauksessa voidaan käyttää sekä staattisia että dynaamisia menetelmiä, joita käsitellään seuraavissa luvuissa.

2.2 Staattinen testaus

Staattisella testaamisella tarkoitetaan ohjelmiston arviointia sisällön perusteella ohjelmaa suorittamatta (IEEE Std 610.12-1990). Staattiseen testaamiseen voidaan laskea myös dokumentaation testaaminen eli sen huolellinen tarkastaminen (Patton 2000, s. 57). Staattinen testaus jaetaan yleensä kahteen osa-alueeseen, *staattiseen analyysiin* ja *katselmointeihin.*

2.2.1 Staattinen analyysi

Staattisen analyysin avulla ohjelmakoodista pyritään löytämään rakenteellisia virheitä ja poikkeamia. Analysointityökaluja on nykyisin saatavilla useita erilaisia sekä ilmaisina että kaupallisina tuotteina eri ympäristöihin. Kattava listaus saatavilla olevista työkaluista löytyy Wikipediasta (2009). Osa työkaluista on integroitu ohjelmointityökalujen sisälle, ja ne suorittavat haluttaessa koodin

analysoinnin jokaisen käännöksen yhteydessä. Sommervillen (2004, s. 527) mukaan kaikki havaitut poikkeamat eivät kuitenkaan aiheuta ohjelmaan suorituksen aikaisia häiriöitä ja ne voivat olla myös tarkoituksellisia. Työkalut antavatkin yleensä poikkeamasta virheen sijaan varoituksen, jonka vakavuuden ihminen voi tarkastaa.

Sommerville (2004, s. 527-528) listaa analyysimenetelmät seuraavasti:

- *Kontrollivirta-analyysi* (suom. TTL 2007, käytetään myös muotoa kontrolli-vuoanalyysi) (engl. control flow analysis)
- *Tietovirta-analyysi* (engl. data use ja information flow analysis)
- *Rajapinta-analyysi* (engl. interface analysis)
- *Polkuanalyysi* (engl. path analysis)

Kontrollivirran analysoinnilla pyritään tutkimaan ohjelman kulkua. Sommervillen mukaan tarkoituksena on löytää erityisesti ohjelmalohkot, joihin ei ole pääsyä sekä mahdollisesti ongelmia aiheuttavat silmukat. Ohjelman kulku voidaan kuvata kaaviona, joka koostuu kolmesta perustoiminnosta: sijoitus, ehto ja iteraatio (Burnstein 2003, s. 102).

Tietovirta-analyysi -termin alle koostuu kaksi Sommervillen mukaan erillistä osa-aluetta. Ensimmäisen (engl. data use) tarkoituksena on havaita rakenteelliset virheet muuttujien käytössä, kuten puuttuvat alustukset sekä käyttämättömät muuttujat. Toinen kokonaisuus (engl. information flow) selvittää muuttujien käyttöä ja keskinäisiä riippuvuuksia ohjelman suorituksen aikana. Tällä menetelmällä voidaan havaita mahdolliset virheet muuttujien sisällön käsittelyssä, esimerkiksi laskentakaavoissa.

Rajapinta-analyysissä tutkitaan parametrien käyttöä funktiokutsuissa. Funktioiden määrittelyjä ja kutsuja tutkimalla pyritään löytämään mahdolliset virheelliset kutsuparametrit, jotka voivat aiheuttaa ohjelman suorituksen aikaisen virheen. Sommervillen (2004) mukaan tämä on erityisen hyödyllistä käytettäessä heikosti tyypitettyjä kieliä kuten C:tä. Kutsuja analysoimalla voidaan myös löytää käyttämättömät funktiot.

Polkuanalyysin avulla pyritään löytämään kaikki mahdolliset reitit, joita ohjelman suoritus voi noudattaa.

Näiden menetelmien lisäksi analyysityökalut voivat tutkia ohjelman rakenteita, turvallisuutta, ohjelmointityyliä (muuttujien nimeäminen, luettavuus) sekä ohjelman monimutkaisuutta ja ylläpidettävyyttä. Nämä tarkastussäännöt pohjautuvat usein kyseiseen ohjelmointikieleen tai -ympäristöön liittyviin standardeihin tai suosituksiin. *Syklomaattinen kompleksisuus* on eräs työkalujen tukema mittari, joka kuvaa ohjelman monimutkaisuutta ilmaisemalla ohjelman sisältämien lineaarisesti riippumattomien polkujen määrän (McCabe 1976). Beizerin (1990) mukaan tätä arvoa voi käyttää arvioimaan haarakattavuuden (ks. luku 2.3.2/Kattavuusmittarit) saavuttamiseksi tarvittavien testitapausten lukumäärää. Jos kompleksisuusarvo kasvaa yli 10:n, tällä on havaittu olevan vaikutusta ohjelman virheherkkyyteen. Beizer ei itse ole täysin vakuuttunut tästä mittarista, ja Kimin (2003) mukaan rivimäärä korreloi voimakkaasti syklomaattisen kompleksisuuden kanssa suurten järjestelmien kohdalla, joten virheherkkyyttä voi arvioida kummalla menetelmällä hyvänsä. Suuri kompleksisuusarvo on kuitenkin merkki siitä, että ohjelman rakennetta tulisi miettiä uudelleen, paitsi jos kyseessä on suoraviivaisesta, mutta laajasta valintarakenteesta johtuva tilanne (Beizer 1990).

2.2.2 Katselmoinnit

Katselmoinnilla (engl. review) tarkoitetaan ryhmässä suoritettavaa ohjelmiston tai siihen liittyvän osan, kuten määrittelydokumentin, arviointia. Katselmusten tarkoituksena on varmistaa, että katselmoitava kohde on määritystensä mukainen. Samalla pyritään löytämään toteutuksesta vikoja ja tarkastetaan, että kohteen toteutuksessa on noudatettu annettuja sääntöjä esimerkiksi ohjelmointityylin osalta. Katselmoinneissa käytetään usein luvussa 2.2.1 mainittuja staattisen analyysin menetelmiä. (Burnstein 2003, s. 307; Patton 2000, s. 95)

Katselmointeja voidaan järjestää eritasoisina alkaen *vertaisarvioinnista* (engl. peer review) virallisiin *tarkastuksiin* (engl. inspection). Vertaisarviointi voi olla hyvinkin epämuodollinen toimenpide, koska sen voi suorittaa kollegan kanssa pyytämällä tätä kommentoimaan omaa työtään. Boehm ja Basili (2001) toteavat, että vertaisarvioinnin avulla vioista havaitaan mediaanina 60 %, joten menetelmää voi pitää tehokkaana. Vertaisarviointia voidaan käyttää muidenkin ohjelmisto-projektin tuotteiden kuin itse koodin tarkastamiseen. Vertaisarviointi on myös hyvä tapa levittää osaamista projektitiimin sisällä, mutta jotta näistä saataisiin selvemmin hyötyä esimerkiksi yrityksen ohjelmistoprosessin parantamiseksi, tulisi vertaisarvioinneistakin kerätä dokumentaatiota ja mittaustietoa (Harjumaa et al. 2006).

Pattonin (2000, s. 98) ja Burnsteinin (2003, s. 308-310) mukaan tarkastukset suoritetaan virallisemmin. Puolueettomuuden varmistamiseksi tarkastuksen vetäjän tulisi olla varsinaisen projektitiimin ulkopuolinen henkilö, ja hänellä tulisi olla koulutus tarkastuksen vetämiseen (Fagan 1999). Vetäjä toimittaa osallistujille ennakkoon tarvittavat materiaalit tutustumista varten. Tilaisuudesta pidetään myös pöytäkirjaa, ja kirjattujen toimenpiteiden suorittamista valvoo tarkastuksen vetäjä.

Kolmas katselmointitapa on ns. *läpikäynnit* (engl. walkthrough). Näissä katselmoitavan kohteen tekijä toimii tilaisuuden vetäjänä ja esittelee ohjelmansa toimintaa rivi riviltä muiden osallistujien esittäessä kysymyksiä ja kritiikkiä. Tilaisuus ei ole kulultaan yhtä virallinen kuin tarkastus. Läpikäynnin osallistujille annetaan yleensä mahdollisuus tutustua katselmoinnin kohteeseen etukäteen. (Burnstein 2003, s. 310-311; Patton 2000, s.97)

Pöytätestaus kuuluu myös katselmointimenetelmiin. IEEE:n standardin (610.12-1990) mukaan myös dokumentoinnin tai testitulosten läpikäyntiä voidaan pitää pöytätestauksena, mutta yleisimmin termillä tarkoitetaan ilman tietokoneen apua suoritettavaa ohjelman toiminnan analysointia (Kaner et al. 1999, s. 47). Usein tämä tarkoittaa rivi riviltä tapahtuvaa koodin läpikäyntiä ja muuttujien tilan, syötteiden ja tulosteiden kirjaamista ylös jokaisen muutoksen yhteydessä. Beizerin (1990, s. 435-437) mukaan pöytätestauksessa ei kannata tehdä mitään sellaista, minkä tietokone voi automaattisesti tarkastaa. Yhtenä esimerkkinä hän mainitsee ohjelman syntaksin tarkastamisen käännettävässä ohjelmassa.

Harjumaa et al. (2006) mukaan katselmoinnit eivät hyödyistään huolimatta ole niin yleisesti käytössä kuin on luultu. Suurimpana esteenä varsinkin pienissä ohjelmistoyrityksissä on resurssipula.

2.3 Dynaaminen testaus

Dynaamisella testauksella tarkoitetaan ohjelmistojärjestelmän tai -komponentin testaamista arvioimalla sen suorituksen aikaista käyttäytymistä (IEEE Std 610.12-1990).

Myers (1979) määritteli ohjelmistotestaamisen avainkysymykseksi: "Millä testitapausten osajoukolla on suurin todennäköisyys havaita eniten virheitä huomioiden annetut aika- ja kustannusrajat?" Testitapausten laatimiselle onkin kehitetty erilaisia menetelmiä, joilla virheiden havaitsemisen todennäköisyyttä voidaan parantaa. Menetelmät eivät ole yksiselitteisiä tapoja, joilla jokainen testaaja voisi luoda aina samat testitapausten joukot, vaan ne ovat enemmän ohjeita, joita testaaja voi oman ammattitaitonsa lisäksi soveltaa parantaakseen testitapausten laatua. Menetelmät täydentävät toisiaan ja niitä tulisikin käyttää rinnakkain parhaan tuloksen saavuttamiseksi. (Myers et al. 2004, s. 44; Burnstein 2003, s. 65).

Dynaaminen testaus jaetaan kahteen osa-alueeseen: *toiminnalliseen* ja *rakenteelliseen* testaukseen, joita käsitellään erikseen seuraavissa alaluvuissa. Testaustavat täydentävät toisiaan virheiden löytämisen tehokkuudessa (Leung & White 1989).

2.3.1 Toiminnallinen testaus

Toiminnallisessa testauksessa testin läpimenoa arvioidaan vertaamalla syötteiden ja suorituksen aikaisten ehtojen avulla saatuja tuloksia toiminnallisten määritysten mukaisiin odotettuihin tuloksiin. Testitapaukset laaditaan järjestelmän tai ohjelmistokomponentin määrittelyjen perusteella tuntematta sen sisäistä rakennetta (IEEE Std 610.12-1990). Tästä syystä testauksesta käytetään myös nimityksiä *määrittelypohjainen* testaus sekä *mustalaatikkotestaus* (engl. black-box testing).

Toiminnallisen testaamiseen kuuluvat myös suorituskykyyn ja käytettävyyteen liittyvien ominaisuuksien testaaminen. Näiden osa-alueiden testimenetelmät on rajattu tämän työn ulkopuolelle.

Toiminnallisten testien suunnittelu voidaan aloittaa projektissa heti, kun toiminnallinen määrittely on testaajien käytettävissä (Burnstein 2003, s.64). Testaajien tulisi osata laatia määrittelyjen perusteella testitapaukset, joilla ohjelman toimivuutta voidaan tutkia. Koska pienenkään ohjelman täydellinen testaaminen ei ole mahdollista johtuen sallittujen syötteiden mahdollisesta määrästä (Patton 2000; s. 38-39), olisi tärkeää, että testaajat osaisivat valita testitapaukset siten, että varmentaminen voidaan tehdä riittävän luotettavalla tavalla rajallisella testitapausten määrällä. Myers et al. (2004) mukaisen käänteisen ajatusmallin mukaan testaajat pyrkivät siis valitsemaan mahdollisimman hyvin virheitä paljastavia testitapauksia.

Toiminnallisten testien riittävyyden tai kattavuuden mittaamiseksi ei ole laadittu selkeitä mittareita. Niiden arvioimiseen käytetään usein rakenteellisen puolen menetelmiä, joista kerrotaan seuraavassa luvussa. Eräs ehdotettu mittari on laskea, kuinka suuren osan vaatimuksista testitapaukset kattavat. (Rajan 2006)

Testitapauksien laatimiseksi on lukuisia menetelmiä. Helpoin tapa on käyttää *satunnaisia syötteitä*. Automaattisilla testityökaluilla ajettuna ja etenkin systeemin rasitustesteissä satunnaisarvojen käyttäminen voi toimia hyvin, mutta pääosin tätä tapaa pidetään tehottomana, koska mikään ei takaa ohjelman kannalta kriittisten arvojen testaamista. Satunnaistestauksella voidaan myös simuloida järjestelmän oikeaa käyttöä, jos arvojen esiintymiselle on olemassa jokin malli (Burnstein 2003, s. 66-67).

Vastaavuusluokittelun avulla pyritään löytämään toisiaan vastaavat syötejoukot, joiden käsittely ohjelmistossa on samanlainen. Tällöin mikä tahansa joukosta

poimittu arvo löytää tai on löytämättä ohjelmassa olevan vian, mikäli luokittelu on tehty oikein. Testaajan tulee huomioida sallittujen arvojen lisäksi myös virheelliset syötearvot luokittelua tehdessään. Näin vikojen löytämisen todennäköisyyttä voidaan kasvattaa pienelläkin määrällä testitapauksia. Vastaavuusluokittelua voidaan tehdä myös odotettujen tuloksien perusteella, mutta tämä on harvinaisempi menettelytapa. Luokitteluun ei ole kiveen hakattuja sääntöjä, vaan se perustuu testaajan osaamiseen ja määrittelyjen tulkitsemiseen. Jos luokittelu on vaikeaa, testaajan on osattava kyseenalaistaa myös määrittelyjen oikeellisuus ja vaatia niihin tarkennuksia. (Burnstein 2003, s. 67-69)

Myers et al. (2004, s. 53-55) ovat luetelleet periaatteet vastaavuusluokittelun tekemiseen ja testitapausten laatimiselle esimerkin kera. Luokat tulee valita siten, että käyvät luokat poimitaan määrittelyn mukaan sallittujen tapausten joukosta ja lisäksi epäkelvot luokat määrittelyn mukaan virheellisistä arvoista. Jos määritys toteaa esimerkiksi sallittujen arvojen olevan välillä 1...5, valitaan yksi testiarvo tältä väliltä sekä epäkelvoiksi tapauksiksi 0 ja 6 rajojen molemmista päistä. Jos on syytä olettaa, että ohjelma ei käsittele laadittujen luokkien jäseniä täsmälleen samalla tavalla, tulee vastaavuusluokka jakaa edelleen pienempiin luokkiin. Kun vastaavuusluokista laaditaan testitapauksia, tulisi positiivisiin testitapauksiin yhdistää aina mahdollisimman monta aiemmin käyttämätöntä käyvän luokan jäsentä. Näin yhdellä testitapauksella voidaan kattaa mahdollisimman laaja kokonaisuus. Epäkelvot syötteet tulee aina testata yksittäin, koska ohjelmat lopettavat usein syötteiden tarkastuksen ensimmäiseen virheeseen.

Vastaavuusluokittelu jättää huomioimatta tiettyjä todennäköisemmin virheitä paljastavia testitapauksia (Myers et al. 2004, s. 59-61). Heidän mukaansa näitä ovat etenkin raja-arvot, joiden tutkimisessa ohjelmoija voi tehdä helposti virheitä. *Raja-arvoanalyysin* avulla testitapauksiin pyritään valitsemaan kelvot syötteet aivan vastaavuusluokkien rajoilta ja epäkelvot syötteet juuri rajojen ulkopuolelta. Määrityksistä voi yrittää löytää myös ohjelman toimintaa koskevia sisäisiä raja-arvoja testitapauksia laajentamaan. Myös tulosteiden joukosta voidaan hakea raja-arvoja, mutta näiden tai virheellisten tulosten aikaansaaminen ei välttämättä onnistu millään syötteillä. Raja-arvoanalyysi on erittäin tehokas tapa virheiden löytämiseen, mutta erityisesti ohjelman sisäisten raja-arvojen tunnistaminen määritysten perusteella vaatii testaajalta taitoa ja kokemusta. Patton (2000, s. 73) listaa tunnistamisen helpottamiseksi seitsemän raja-arvojen tyyppiä: numeeriset, merkistölliset, määrään, nopeuteen, kokoon, paikkaan tai sijaintiin (engl. position/location) liittyvät seikat.

Vastaavuusluokittelun ja raja-arvoanalyysin ongelmana on se, ettei niiden avulla voi tutkia syötteiden yhteisvaikutuksia. Testien kattavuuden varmistamiseksi suositellaan muiden menetelmien samanaikaista käyttöä sekä toiminnallisen että rakenteellisen testauksen puolelta. (Burnstein 2003, s. 77-78, Myers et al. 2004, s. 65-66)

Syy-seurauskaavion laatiminen mahdollistaa ehtojen yhdistelyn. Ideana on purkaa määrittely ensin pienempiin yksiköihin, jotta kaavio pysyy hallitun kokoisena. Näille yksiköille testaaja pyrkii tunnistamaan eri syötteiden (syiden) ja tulosteiden tai ohjelman tilamuutosten (seurauksien) väliset riippuvuudet, joista laaditaan Boolen operaattoreita JA, EI ja TAI käyttäen looginen kaavio. Kaavioon voidaan kuvata myös määrityksistä seuraavia rajoituksia ehtojen välillä. Kaavion perusteella laaditaan päätöstaulu, josta muodostetaan lopulliset testitapaukset. Menetelmän etuina pidetään sekä ehtoja yhdistävien testitapausten systemaattisen

muodostamisen että määrityksissä olevien puutteiden ja epäselvyyksien havaitsemisen mahdollisuutta. (Myers et al. 2004, s. 67-88)

Tilasiirtymien testaaminen mahdollistaa erilaisen lähestymistavan testien laatimiseen, mikäli määrittelydokumentaatio sisältää tilakaavion tai -taulukon järjestelmän toiminnasta. Tilojen lisäksi kaaviosta tulisi käydä ilmi syötteet ja tapahtumat, jotka käynnistävät tilasiirtymät sekä siirtymien aiheuttamat tulosteet tai toiminnot. Kaavion avulla testaaja voi varmistaa, että järjestelmä käy jokaisessa tilassaan. Kattavampaa testausta haluttaessa voidaan testata jokainen siirtymä erikseen. Näin eri tilojen toimivuus voidaan varmistaa kohtuullisen luotettavasti, vaikkei kaikkia eri *polkuja* (ks. luku 2.3.2) ohjelman sisällä olisikaan mahdollista testata erikseen. Tilasiirtymien testaaminen voi kuitenkin vaatia erikoistyökalujen tai testattavan ohjelman erillisen testikäännöksen käyttöä, jotta testaaja pääsee näkemään tarvittavat tilasiirtymiin liittyvät muuttujat ja tapahtumat. (Burnstein 2003, s. 82-85)

Näiden menetelmien lisäksi testaaja voi laatia testitapauksia *valistuneesti arvaamalla* eli testaamalla tekijöitä, joilla on aiemmin havainnut olevan yhteyttä samantyyppisissä toiminnoissa havaittuihin vikoihin. Tätä testaustapaa kutsutaan myös *kokeilevaksi testaukseksi* (engl. exploratory testing; Kaner et al. 1999, s. 6). Esimerkkeinä tästä on nollan käyttö syötteenä mahdollisen nollalla jakamisen havaitsemiseksi, listan täyttäminen identtisillä arvoilla, listojen koon vaihtelu jne. (Burnstein 2003, s. 85-86; Myers et al. 2004, s. 88-89).

Myers et al. (2004, s. 90) esittelee viisiaskelisen kokonaisstrategian testitapausten laatimiselle edellisiin menetelmiin perustuen. Tämäkään tapa ei takaa kaiken kattavaa testitapausten joukkoa, mutta tekijöiden mukaan systemaattinen lähestyminen antaa hyvän todennäköisyyden laadukkaiden testien suunnittelulle. Myös Leung ja White (1989) suosittelevat samanlaista järjestystä toiminnallisista rakenteellisiin menetelmiin.

1. Mikäli määrittelyt sisältävät ehtojen yhdistelmiä, aloita syy-seuraus-kaavion laatimisella.
2. Käytä aina raja-arvoanalyysiä sekä syötteille että tulosteille.
3. Laadi vastaavuusluokittelu syötteille ja tulosteille.
4. Laadi lisätestejä arvausmenetelmällä.
5. Jatka testien laatimista rakenteellisen testaamisen menetelmillä.

2.3.2 Rakenteellinen testaus

Rakenteellisessa testauksessa testitapaukset laaditaan järjestelmän tai ohjelmistokomponentin sisäistä rakennetta analysoimalla (IEEE Std 610.12-1990). Rakenteellista testausta kutsutaan myös *valko-* ja *lasilaatikkotestaukseksi* (TTL 2007; engl. white-box testing), koska näissä nähdään testattavan kohteen sisälle, toisin kuin mustalaatikkotestauksessa. Koska rakenteellinen testaus vaatii ohjelmakoodiin tai pseudokooditasolla oleviin teknisiin suunnitelmiin tutustumista, ajoittuu rakenteellisten testien suunnittelu yleensä myöhempään vaiheeseen kuin toiminnallisten testien suunnittelu (Burnstein 2003, s.97-99). Samasta syystä rakenteelliset testit laaditaan yleensä kehittäjien toimesta ja ne soveltuvatkin luontevimmin yksikkötestaukseen (Weyuker 1998).

Rakenteellisen analyysin avulla testit pyritään saamaan mahdollisimman *kattaviksi*, eli suorittamaan mahdollisimman suuri osa ohjelman rakenteista, mutta samalla minimoimaan tähän tarvittavien testitapausten määrä. Etenkin valmiin koodin analysoinnissa käytetään luvussa 2.2.1 kuvattuja staattisia menetelmiä.

Kattavuusmittarit

Testien kattavuuden mittaamiseen on lukuisia eritasoisia mittareita, joita käyttäen testaussuunnitelmassa voidaan ottaa suoraan kantaa halutun testaustason saavuttamiseen ja kiinnittää mittari testaamisen lopetuskriteeriksi (Burnstein 2003, s. 99-101).

Yksinkertaisin mittari on *lausekattavuus* (engl. statement coverage). Tällä ilmaistaan prosenttiosuutena se määrä ohjelman riveistä, jotka valituilla testitapauksilla on saatu suoritettua. Jokaisen koodirivin suorittaminen testien avulla tarkoittaa siis 100 % lausekattavuutta. Lausekattavuus on heikko kriteeri, koska 99 % kattavuus voi tarkoittaa sitä, että kriittisin ohjelmakohta on jätetty testaamatta, eikä edes täysi lausekattavuus kerro sitä, kuinka hyvin ohjelmakoodin sisältämät ehdot on testattu (Myers et al. 2004, s.45; Patton 2000, s. 122). Lausekattavuutta yleisemmin mittarina käytetään *lohkokattavuutta* (engl. basic block coverage), joka yhdistää haarautumattomat peräkkäiset lauseet yhdeksi lohkoksi ja mittaa lohkojen läpikäyntiä rivien sijaan. Lohkokattavuutta käytetään yleisesti mittarina käytännön tutkimuksissa (Gittens et al. 2002).

Päätös- tai *haarakattavuus* (engl. decision tai branch coverage) mittaa, kuinka suuri osa koodin sisältämistä päätöksistä tai päätösten perusteella valituista ohjelmakoodin haaroista on käyty läpi testitapauksien avulla. Täyteen 100 % kattavuuteen päästään näillä mittareilla, kun jokainen ohjelmakoodin sisältämä ehtojen pohjalta tehtävä päätös saadaan testattua kaikilla eri tavoin käsiteltävillä lopputuloksilla – toisin sanoen jokaisen ehtolauseen kohdalla testitapauksen tulee suorittaa sekä tosi että epätosi vaihtoehto ja valintalauseiden (switch) kohdalla kaikki vaihtoehtoiset lohkot. Erikoistilanteita lukuun ottamatta täysi päätös- tai haarakattavuus takaa myös täyden lausekattavuuden. (Burnstein 2003, s. 103-104; Myers et al. 2004, s. 45-46). Kimin (2003) mukaan lohko- ja haarakattavuus korreloivat voimakkaasti keskenään, joten testauskriteeriksi voidaan valita näistä kumpi hyvänsä, kunhan valinta huomioidaan virheitä arvioidessa.

Ehtokattavuus (engl. condition coverage) mittaa prosenttiosuutta sille, kuinka suuren osan ehtojen tuloksista testitapaukset ovat käyneet läpi. Täysi kattavuus edellyttää jokaisen päätöslausekkeen sisältämien yksittäisten ehtojen testaamista eri tulosvaihtoehdoilla; esimerkiksi kaksi erillistä ehtoa sisältävä ehtolause pitää testata siten, että molemmat ehdot saavat sekä toden että epätoden arvon. Täysi ehtokattavuus ei kuitenkaan takaa täyttä päätöskattavuutta. Ehtokattavuuden sijaan käytetäänkin usein *päätösehtokattavuutta* (engl. decision/condition coverage), jolla varmistetaan myös päätösten ja ohjelmahaarojen testaaminen eri lopputuloksilla, mikäli tavoitellaan 100 % kattavuutta. (Burnstein 2003, s.105-107; Myers et al. 2004, s. 46-49).

Täysi päätösehtokattavuuskaan ei ole riittävä kriteeri sille, että kaikki yksittäiset ehdot on testattu kattavasti. Tähän päästään vain täydellä *moniehtokattavuudella* (engl. multiple condition coverage). Tällöin testitapausten tulee testata jokaisen päätöslauseen sisältämien yksittäisten ehtojen kaikki yhdistelmät. Moniehto-kattavuudenkin kohdalla testitapaukset voivat testata erillisten päätöslauseiden ehtoja, jolloin tarvittavien tapausten määrää voidaan yhä tiivistää. (Burnstein 2003, s. 108; Myers et al. 2004, s. 49-52)

Edellä lueteltujen kontrollivirtaan perustuvien kattavuusmittareiden voima eli testauksen yksityiskohtaisuus kasvaa luetellussa järjestyksessä – lausekattavuus on tehokkuusmittarina heikoin ja moniehtokattavuus vahvin. Voimakkaampaa kattavuusmittaria käytettäessä korkean prosentin saaminen vaatii yleensä

suurempaa panosta testausresursseihin. Mittarikriteerejä asetettaessa on mietittävä, mikä on kyseiselle projektille ohjelman kriittisyyden ja käytettävien resurssien puitteissa järkevä kattavuustavoite. Tavoitteita määritettäessä tulee myös huomioida, että täysi kattavuus on harvoin saavutettavissa (Piwowarski et al. 1993). Ohjelmakoodi voi sisältää paikkoja, joihin ei ole edes mahdollista päästä, eikä niitä voi näin ollen myöskään testata. (Burnstein 2003, s. 106-108 & 111)

Kattavuuden mittaamiseksi ohjelmakoodi pitää *instrumentoida* eli varustaa tarvittavilla *antureilla* (engl. probe), jotka rekisteröivät ja raportoivat mittaustyökaluille testien läpikäymät ohjelmahaarat tai vertailut. Instrumentointi ja mittaus voi hoitua joko suoraan kehitystyökaluihin integroiduilla tai erillisillä testaustyökaluilla. Haittapuolena mittausten tekemiselle on, että instrumentointi muokkaa ohjelmakoodia ja voi aiheuttaa muutoksia sen toimintaan ja varsinkin suorituskykyyn. (Kaner et al. 1999, s.200-201)

Kontrollivirtaan perustuvien menetelmien lisäksi testitapauksia voidaan suunnitella tietovirta-analyysin avulla. Tällä tavoin voidaan laatia osittaisia polkuja ohjelman läpi. Kattavin tietovirta-analyysiin pohjautuva menetelmä on laatia testitapaukset siten, että ohjelmakoodista tulee suoritettua jokainen polku jokaisen muuttujan asettamisesta sen käyttämiseen eli niin kutsuttu *määrittely-käyttöpolku* (engl. def-use path).

Kaikkein voimakkain kattavuuskriteeri on *polkukattavuus* (engl. path coverage, Burnstein 2003, s. 119). Tällä mitataan testijoukon suorittamien erilaisten polkujen osuutta kaikista mahdollisista poluista, jota ohjelman suoritus voi noudattaa. Polkujen lukumäärä voi olla hyvinkin yksinkertaisessa ohjelmassa jo niin suuri, ettei täyden polkukattavuuden saavuttaminen ole mahdollista tai ainakaan käytännöllistä.

Silmukoiden testaaminen

Ohjelman sisältämät *silmukat* ovat herkkiä vioille (Burnstein 2003, s. 115). Yksinkertaisten silmukoiden testaukseen Beizer (1990, s. 81-85) suosittelee laadittavaksi testitapaukset siten, että silmukan suoritusta testataan sekä eri iteraatiomäärillä että silmukkamuuttujan arvoilla, ja niille haetaan testitapauksia raja-arvoanalyysin avulla sekä ala- että ylärajoilta. Tämän lisäksi Beizer suosittelee testaamaan silmukkaa jollain tyypillisellä sallitulla arvolla.

Beizer kuvaa myös menetelmän, jolla toisistaan riippuvien sisäkkäisten ja peräkkäisten silmukoiden testaus voidaan suunnitella niin, ettei testitapausten määrä kasva räjähdysmäisesti. Tilannetta yksinkertaistetaan siten, että kerrallaan tarkastellaan vain yhtä silmukkaa sisimmäisestä alkaen ja vakioidaan muiden tilanne. Tämä on aina kompromissi testien kattavuuden ja resurssien kannalta järkevän testaamisen välillä. Monimutkaisten silmukkarakenteiden kohdalla Beizer suosittelee sekä määritysten että koodin oikeellisuuden varmistamista ennen testitapausten laatimista.

Vertailutestaus

Burnstein (2003, s.116-118) lisää rakenteelliseen testaamiseen myös *vertailutestauksen* (engl. mutation testing ja back-to-back testing), jonka tarkoituksena on osoittaa tietyntyyppisten virheiden olemassaolo epätodennäköiseksi. Vertailutestauksessa alkuperäiseen ohjelmakoodiin luodaan tarkoituksellisia, yleensä yksinkertaisia virheitä. Tällaisia ovat esimerkiksi vakioarvojen muutokset, vertailu- ja laskuoperaattorien vaihdot, lauseiden poisto

jne. Näitä muokattuja ohjelmaversioita kutsutaan *mutanteiksi*. Mutanttien luomiseen on olemassa valmiita ohjelmia, jolloin mutanttien määrä saadaan helposti suureksi.

Mutantti testataan samoilla testitapauksilla kuin alkuperäinenkin ohjelma. Hyvä testitapaus antaa mutantin kohdalla alkuperäisestä poikkeavan testituloksen eli havaitsee virheen. Mikäli virhettä ei havaita, tulee tilanne analysoida ja laatia testitapaus, jolla mutantin virhe havaitaan. Poikkeuksena tästä ovat tilanteet, jossa mutantti on yhtenevä alkuperäisen ohjelman kanssa. Kun testitapaus havaitsee mutantin virheen, on todennäköistä, että se havaitsee muutkin vastaavat virheet, jolloin luottamus testitapauksen laatuun paranee. Vertailutestaus on työlästä, joten sitä käytetään yleisimmin yksikkötestauksen apuna.

2.3.3 Testauksen lopettaminen ja kustannustehokkuus

Testaukselle tulisi asettaa selkeät lopetuskriteerit, joiden täyttyessä on saavutettu etukäteen tarpeelliseksi katsottu testauksen taso. Kriteereissä olisi hyvä olla mukana valittuun kattavuusmittariin kytketty arvo (Burnstein 2003, s. 99-101).

Piwowarski et al. (1993) toteavat, että 100 % lausekattavuustavoite voi olla mahdollista saavuttaa yksikkötestausvaiheessa (ks. luku 2.4.2), mutta *järjestelmätestausvaiheen* (ks. luku 2.4.4) toiminnallisissa testeissä 70-80 % lausekattavuus on tavoiteltava arvo, jolla ohjelmistosta karsiutuu pois suurin osa asiakkaille näkyvistä virheistä. Tämän jälkeen kattavuuden kasvattaminen ei vaikuta enää yhtä tehokkaasti käyttäjien havaitsemien virheiden määrän vähentymiseen. Gittens et al. (2002) raportoivat samanlaisista tuloksista.

Kim (2003) on havainnut, että suurissa järjestelmissä tarkka kattavuusmittaus ei ole tarpeellista eikä tehokasta, sillä virheet keskittyvät tiettyihin moduuleihin: 20 % moduuleista aiheuttaa 70 % vioista. Käytännössä suurten järjestelmien kohdalla tarkkojen kattavuusmittausten toteutus on haastavaa, sillä koodin instrumentointi ja siitä aiheutuvat suorituskykyhaitat vaikeuttavat testaamista ja jopa peittävät virheitä, jotka riippuvat toimintojen ajoituksesta.

Kim (2003) ehdottaakin, että virheherkät moduulit tulisi tunnistaa virhejakauman perusteella, etenkin jos ohjelmistosta on olemassa aiempia versioita ja testaushistoriaa. Kattavuusanalyysiin Kim ehdottaa helpommin toteutettavaa mallia, jossa kattavuutta mitataan pääosin funktiotasolla. Lohko- tai päätöskattavuuden tasolle on syytä mennä vain virheherkiksi tunnistetuissa moduuleissa. Myös kattavuustavoite tulisi asettaa korkeammaksi virheherkkyyden kasvaessa. Gittens et al. (2006) ehdottavat testien priorisointia ja korkeamman kattavuustavoitteen asettamista niille toiminnallisille testeille, jotka ovat joko helppoja testata järjestelmätestausvaiheessa tai ovat kriittisiä ohjelmiston toimivuuden kannalta. Vähemmän tärkeät tai vaikeasti testattavat osat voidaan jättää matalammalle kattavuustavoitteelle ja osin jopa pelkästään yksikkötestauksen ja katselmointien varaan.

2.4 Testaustasot ohjelmistokehityksessä

2.4.1 Ohjelmistokehityksen mallit

Ohjelmistokehityksen perinteisin malli on ns. vesiputousmalli, jota kutsutaan myös ohjelmiston elinkaareksi (Sommerville 2004, s. 66-68). Puhtaassa vesiputousmallissa ohjelmisto toteutetaan alusta loppuun erillisissä vaiheissa aloittaen vaatimusten ja toiminnallisuuksien määrittelyistä, joiden pohjalta ohjelmisto suunnitellaan ja vasta sen jälkeen toteutetaan. Toteutusvaiheen

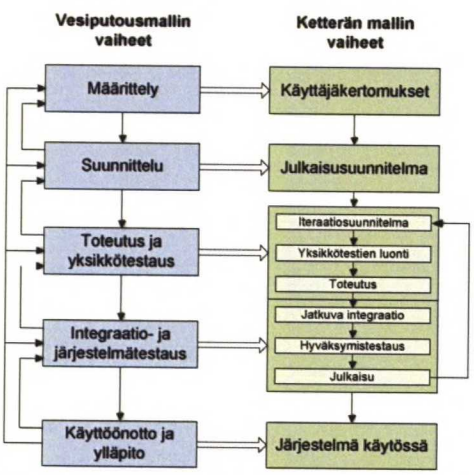
yksikkötestausta lukuun ottamatta testaus on vesiputousmallissa oma vaiheensa, joka suoritetaan vasta toteutuksen jälkeen. Testauksen jälkeen ohjelmisto otetaan käyttöön ja se siirtyy ylläpitoon.

Inkrementaalissa kehitystavassa ohjelmiston kehitys tapahtuu vaiheittain useana osatoimituksena. Inkrementaalinen malli voi koostua useasta peräkkäisestä vesiputouksesta, jolloin uusi vaihe määritellään aina ennen sen toimitusta. Vaihtoehtoisesti kokonaisuus on voitu määritellä ja suunnitella valmiiksi jo alussa, mutta toteutus-, testaus- ja käyttöönotto on jaettu osatoimituksiin. (Poimala et al. 2008)

Iteratiivisessa kehitystavassa ohjelmistosta rakennetaan nopeasti ensimmäinen, usein toiminnoiltaan karsittu versio, jonka kehittämistä jatketaan vaiheittain, kunnes asiakas hyväksyy ohjelmiston. Iteratiivisen mallin etuna on, että asiakas pääsee näkemään ohjelmiston aikaisessa vaiheessa ja mahdolliset väärinkäsitykset määrittelyissä voidaan havaita ennen hyväksymistestausta. Haasteena puolestaan on se, että suunnittelun vähäisyyden vuoksi ohjelman rakenne voi muodostua huonoksi uusien toimintojen ja ylläpidon kannalta. (Poimala et al. 2008)

Ketterät kehitystavat yhdistävät inkrementaalisen ja iteratiivisen mallin lisäten prosessin sisään ohjelmiston laatua parantavia toimintoja, kuten *refaktorointi* ja *jatkuva integraatio.* Refaktoroinnilla tarkoitetaan koodin muuttamista ja siistimistä ilman että sen toiminnallisuus muuttuu (Opdyke 1992). Jatkuvalla integraatiolla taas tarkoitetaan sitä, että ohjelmistoon toteutettavat lisäykset pyritään tekemään niin, että kokonaisuus pysyy jatkuvasti toimivana. Muutosten ja lisäysten jälkeisen toimivuuden varmistaminen vaatii jatkuvaa testausta, jolloin myös testiautomaation hyödyllisyys kasvaa. (Myers et al. 2004, s. 184; Poimala et al. 2008)

Käytettävän kehitysmalli voidaan valita projektikohtaisesti ja niitä voidaan tarvittaessa yhdistellä saman projektin sisällä (Sommerville 2004; Boehm 2002; DeMarco & Boehm 2002). Huo et al. (2004) toteavat ketterien kehitysmallien eduksi myös sen, että niissä laadunvarmistukseen liittyviä tehtäviä on mahdollista toteuttaa aiemmin kuin perinteisessä vesiputousmallissa, koska itse asiassa vesiputousmallin sisältämät vaiheet nivoutuvat jokaisen iteraation sisään. Näin ollen testausmenetelmiä voidaan käyttää projektityypistä riippumatta. Kuvassa 2.1 on havainnollistettu sekä vesiputousmallin että ketterien mallien päävaiheet ja niiden vastaavuudet Sommervillen (2004, kuva 4.1) ja Huo et al. (2004, kuva 1) mukaisesti.

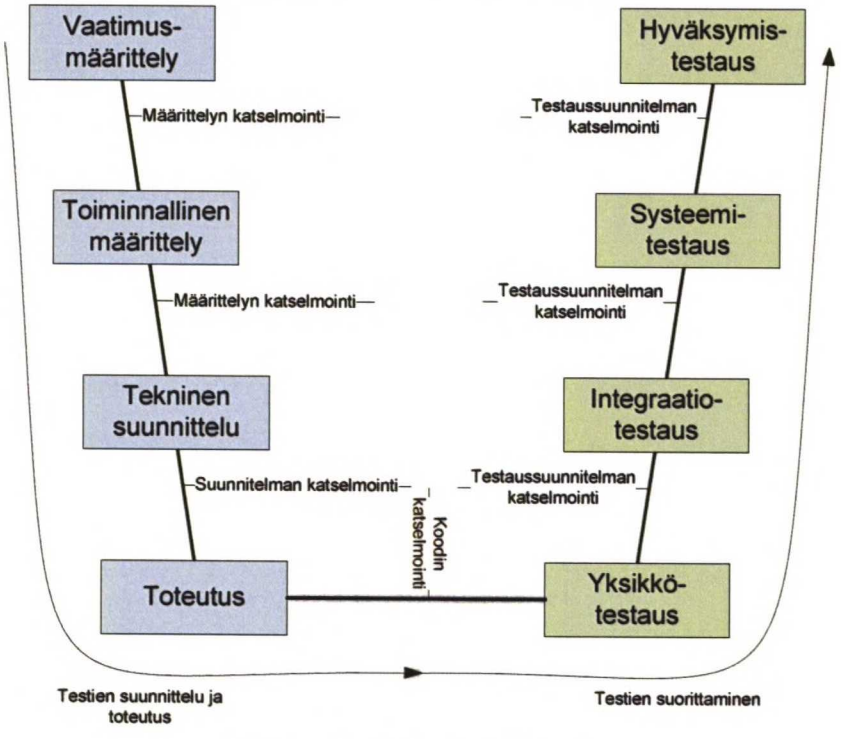


Kuva 2.1: Vesiputousmallin ja ketterien mallien vertailu

Kuvassa 2.2 on esitetty Burnsteinin (2003, kuva 1.6) esitykseen perustuva testauksen laajennettu V-malli, jossa kuvataan ohjelmistokehitysvaiheiden ja testaustasojen väliset suhteet katselmointipisteineen. Kunkin testaustason testien suunnittelun pitäisi pohjautua vasemman reunan vaiheessa tehtyihin määrityksiin (Burnstein 2003; Myers et al. 2004). Myers et al. tosin erottavat toiminnallista määrittelyä vastaavaksi testivaiheeksi toiminnallisen testauksen ja liittää systeemitestauksen kokonaisjärjestelmälle asetettujen muiden tavoitteiden, kuten suorituskyvyn, testaamiseen. Tässä työssä systeemitestauksen katsotaan kattavan molempien osa-alueiden testaamisen.

Burnstein (2003) on lisännyt V-malliin määrittelyjen katselmoinnit tarkistuspisteiksi, joiden jälkeen vastaavan testivaiheen suunnittelu voidaan aloittaa. Malli soveltuu myös ketteriin menetelmiin, koska vastaavat vaiheet on tunnistettavissa jokaisen iteraation sisällä edellä todetulla tavalla. Jatkuvan integraation vuoksi yksikkö- ja integraatiotestauksen pitäisi olla säännöllistä, ja systeemi- ja hyväksymistestaus tapahtuu iteraation loppuvaiheessa samoin kuin vesiputousmallin testausvaiheessa.

Seuraavissa alaluvuissa käsitellään tarkemmin eri testaustasoja.



Kuva 2.2: Laajennettu V-malli

2.4.2 Yksikkötestaus

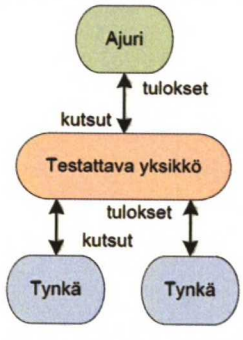
Yksikkötestaus tarkoittaa yksittäisten tai toisiinsa liittyvien yksiköiden testaamista (IEEE Std 610.12-1990). Burnstein (2003, s. 137) ja IEEE (Std 610.12-1990) määrittelevät *yksikön* ohjelmiston pienimmäksi mahdolliseksi testattavaksi komponentiksi. Yksikkö voi olla yksittäinen funktio, luokka tai luokan metodi, jokin komponentti tai sen yksittäinen toiminnallisuus, joka voidaan testata erillään muusta ohjelmasta riippumatta (Burnstein 2003; Sommerville 2004, s. 548). Myös Hamlet (2006) korostaa yksikön määritelmässä sen täydellistä itsenäisyyttä. Tämä vaatii ohjelmiston testattavuuden huomioimista jo suunnitteluvaiheessa.

Jälkikäteen testattavuuden parantaminen riippuvuuksia poistamalla voi vaatia suuria muutoksia ohjelmakoodiin (Wirfs-Brock 2009).

Burnstein (2003) toteaa, että olio-ohjelmoinnissa yksikkötason valintaan liittyy aina kompromissi. Yksittäisen metodin tasolle mentäessä voidaan toiminnallisuus varmistaa paremmin, mutta tällöin luokan sisäiset riippuvuudet sen muista metodeista tulisi kiertää testikehyksen avulla, mikä voi osoittautua suuritöiseksi. Mikäli luokkaa testataan yksikkönä, helpottuu testien suorittaminen ns. *testikehyksen* kautta, mutta toisaalta testitapausten laatiminen ja testauksen kattavuuden arviointi tulee vaikeammaksi. Luokkatason yksikkötestaus vaatii myös luokan sisäisten tilamuutosten huomioimisen, mikä monimutkaistaa testausprosessia.

Myers et al. (2004, s. 91) mukaan yksikkötestaukselle on kolme merkittävää syytä. Testaaminen voidaan tehdä hallitummin, kun testataan vain pientä osaa kerrallaan. Toiseksi yksikkötestaus helpottaa vikojen löytämistä ja korjaamista, koska virheen sijainti tiedetään kohtuullisen tarkasti testattavan kohteen pienestä koosta johtuen. Kolmantena etuna mainitaan mahdollisuus usean moduulin rinnakkaiseen testaamiseen.

Yksikkötestauksen suorittamiseksi joudutaan yleensä rakentamaan kuvan 2.3 kaltainen testikehys (Burnstein 2003, s. 148-150). Testattavaa yksikköä kutsutaan *ajurilla* (engl. driver), joka välittää testitapausten mukaiset kutsut ja parametrit testattavalle yksikölle ja ottaa vastaan kutsujen tulokset. Mikäli testattavan yksikön tarvitsee kutsua itsensä ulkopuolisia palveluja, on näitä varten toteutettava ns. *tynkäpalveluja* tai *-luokkia* (engl. stub). Tyngät matkivat oikeiden palveluiden toimintaa. Näin yksikköä voidaan testata, vaikka kutsuttavaa palvelua ei olisi vielä olemassakaan. Samalla voidaan huolehtia siitä, että testitapaukset ajetaan vakioiduilla arvoilla, mikä ei välttämättä ole mahdollista oikeaa rajapintaa käytettäessä. Myers et al. (2004, s. 105-119) pitävät myös mahdollisena aiemmin yksikkötestatun komponentin käyttämistä ajurina tai tynkänä, mitä voidaan kuitenkin pitää jo integraatiotestauksena (ks. luku 2.4.3).



Kuva 2.3: Testikehys

Ketterässä kehityksessä, esimerkiksi *extreme programming* -mallissa, yksikkötestaus on keskeisimpiä toimenpiteitä, ja nämä testit tulisi laatia aina ennen toteutuksen aloittamista (Beck 1999). Tätä kutsutaan myös *testauslähtöiseksi* kehitystavaksi (TTL 2007, engl. test driven development). Ennen toteutusta testitapausten laatimiseen voidaan käyttää kuitenkin vain toiminnallisen testauksen menetelmiä. Rakenteellisten menetelmien käyttö voidaan aloittaa vasta, kun käytettävissä on varsinaista ohjelmakoodia. Koodimuutokset voivat vaatia myös rakenteellisten testien päivittämistä uutta koodia vastaaviksi (Leung & White 1989).

Yksikkötestauksen suunnittelua ja suorittamista ohjeistamaan on laadittu IEEE:n standardi (Std 1008-1987). Suunnitteluvaiheessa laaditaan normaalin projektisuunnitelman osat eli resursointi, käytettävät tai toteutettavat työkalut, aikataulu ja riskianalyysi. Lisäksi valitaan lähestymistapa testaamiselle: testitapausten suunnittelumenetelmät, kattavuustavoitteet ja -mittarit sekä ominaisuudet, joita testataan. Myös testien päättämiskriteerit tulee määritellä sekä hyväksyttäville että hylkääville vaihtoehdoille. Suunnittelun jälkeen laaditaan testitapaukset ja suoritetaan testaus suunnitelman mukaan. Suorituksen jälkeen tulokset analysoidaan, raportoidaan ja päätetään testien jatkamisesta tai lopettamisesta.

Yksikkötestien suunnittelussa Myers et al. (2004, s. 92) pitää rakenteellista lähestymistapaa tärkeimpänä menetelmänä. Näin saatuja testitapauksia voidaan laajentaa määrittelyihin pohjautuvilla toiminnallisen testauksen menetelmillä. Burnstein (2003, s. 142) on samaa mieltä, mutta huomauttaa, että valmiskomponentteja käytettäessä toiminnallinen testaaminen voi olla ainoa vaihtoehto. Burnsteinin mielestä kriittisten toimintojen kohdalla yksikkötesteihin tulisi sisällyttää myös suorituskyky-, kuormitus- ja tietoturvatestaus.

2.4.3 Integraatiotestaus

Integraatiotestauksella on Burnsteinin (2003, s. 152) mukaan kaksi tavoitetta: havaita yksiköiden välisissä rajapinnoissa olevia vikoja ja samanaikaisesti koostaa erikseen testatuista yksiköistä toimiva kokonaisuus järjestelmätestausta varten. Integraatiotestiin tulisi ottaa vain yksikkötestit läpäisseitä komponentteja. IEEE:n (Std 610.12-1990) määritelmän mukaan integraatiotestauksessa yhdistetään ohjelmisto ja/tai laitteistokomponentteja ja arvioidaan niiden välisiä vuorovaikutuksia.

Myers et al. (2004, s. 105-119) eivät varsinaisesti erittele integraatiotestausta yksikkötestauksesta, vaan esittelevät kaksi mallia yksikkötestaukselle: inkrementaalinen ja ei-inkrementaalinen ("big bang"). Jälkimmäisessä kaikki yksiköt yksikkötestataan erikseen ja yhdistetään kerralla kokonaisuudeksi. Inkrementaalisessa mallissa järjestelmä koostetaan lisäämällä testattuja yksiköitä ketjuun yksi kerrallaan. Ajurien ja tynkien avulla varmistetaan, että koostettu kokonaisuus toimii ennen seuraavan kerroksen lisäämistä. Myersin mielestä inkrementaalinen malli on ylivoimainen: se vähentää ajurien ja tynkien kirjoittamiseen vaadittavaa työmäärää, aikaistaa ja helpottaa rajapintoihin liittyvien virheiden löytämistä ja tuottaa perusteellisemman testauksen, kun ensin integroidut yksiköt ovat mukana myös seuraavien kierrosten testeissä. Burnstein (2003, s.153) pitää Myersin inkrementaalista mallia integraatiotestauksena ja sen oikeana suoritustapana.

Inkrementaalinen koostaminen voidaan tehdä sekä ylhäältä alas että alhaalta ylös. Alhaalta ylös tultaessa yksikkötestataan ensin alimman tason yksiköt, minkä jälkeen integroidaan niitä kutsuvat yksiköt. Testaus tapahtuu kutsumalla ylimmän käytössä olevan tason yksikköä ajurin avulla. Onnistuneen testin jälkeen lisätään taas seuraava kerros edellisten päälle. Ylhäältä alas tullessa testataan ensimmäiseksi ylimmän tason päämoduuli, jonka käyttämät alemman tason yksiköt korvataan tyngillä. Onnistuneen testin jälkeen tyngät korvataan seuraavan tason yksiköillä ja niiden tarvitsemat alemman tason yksiköt korvataan taas tyngillä, kunnes lopulta koko ketju on integroitu.

Myers et al. (2004, s.118-119) mukaan molemmilla tavoilla on hyvät ja huonot puolensa. Virheet paljastuvat nopeimmin aloitussuunnasta, mikä kannattaa huomioida, jos epäillään vakavia tai useita virheitä nimenomaan toisessa päässä

ketjua olevissa yksiköissä. Ylhäältä alas tullessa suuri etu on se, että ohjelmasta saadaan toimiva runko kasaan jo alkuvaiheessa ja sitä voidaan käyttää esittelyihin. Tällä on myös kehittäjäryhmää motivoiva vaikutus. Suurimpana haittana Myers et al. pitävät tynkien kirjoittamista, mikä ei ole välttämättä helppo tehtävä. Alhaalta ylös tultaessa integroiminen on helpompaa, koska ajureita tarvitaan yleensä vähemmän kuin tynkiä. Myös testitapausten laatiminen alhaalta ylös tultaessa on helpompaa. Suurimpana haittana tässä mallissa he pitävät sitä, että ohjelmasta ei saada toimivaa versiota ennen kuin kaikki yksiköt on integroitu kasaan. Kaner et al. (1999, s. 46) ovat havainneet, että käytännössä yksiköiden valmistumisjärjestys ratkaisee integrointijärjestyksen.

Burnstein (2003, s. 158) huomauttaa, että inkrementaalinen malli ei toimi sellaisenaan oliosuuntautuneiden ohjelmien testaamisessa, koska niissä inkrementaalisen mallin vaatima hierarkia ei ole välttämättä selvä. Burnstein suosittelee vastaavan idean toteuttamista luokkia ryhmittelemällä. Vaihtoehtoja on kaksi. Ensimmäisessä luokat ryhmitellään siten, että alhaisen tason toimintoja toteuttavat toisiaan käyttävät luokat kerätään ryhmäksi, joka testataan tarvittavien ajurien ja tynkien avulla. Näitä yhdistellään keskenään testauksen edetessä, kunnes koko järjestelmä on integroitu. Vaihtoehtoisessa tavassa luokat valitaan siten, että ensin testataan luokat, joilla on joko hyvin vähän tai ei lainkaan liittymiä muihin luokkiin tai palveluihin. Näiden testaamisen jälkeen voidaan integroida testattuja luokkia käyttävät luokat, ja tätä toistetaan uudelleen, kunnes koko järjestelmä on koossa.

Labiche et al. (2000) ja Kung et al. (1993) ovat sitä mieltä, että tynkien määrä tulisi pitää mahdollisimman pienenä, koska luotettavien ja normaalikoodia yksinkertaisempien tynkien laatiminen ei ole aina mahdollista, niiden laatimista ei voida kaikissa tilanteissa automatisoida ja ne voivat olla jopa herkempiä vioille kuin varsinainen ohjelma (Beizer 1990). Tämä on ymmärrettävää, koska olio-ohjelmissa suuri osa luokkien metodeista on hyvin pieniä, muutaman rivin mittaisia (Wilde & Huitt 1992). Labiche et al. ovat laatineet myös mallin ja työkalun, joilla oliosuuntautuneen ohjelman luokkien testausjärjestystä ja tarvittavien tynkien määrää voidaan optimoida jo ohjelman suunnitteluvaiheessa luokkamallin avulla.

Burnstein (2003, s. 159-162) suosittelee integraatiotestien laatimiseen yksikkötestien tavoin sekä rakenteellisia että toiminnallisia menetelmiä. Osaa yksikkötesteistä voi olla mahdollista käyttää integraatiotestauksessakin, mutta on tärkeää laatia uusia testitapauksia, jotka huomioivat yksiköiden väliset liittymät. Uusien testien tulee erityisesti varmistaa, että kaikkia integroituja yksiköitä kutsutaan vähintään kerran ja että eri lähteistä suoritettavat kutsut tulee testattua mahdollisimman kattavasti. Olio-ohjelmia testattaessa tulee huomioida myös luokkien välisten perintäsuhteiden ja abstraktioiden vaikutus testien kattavuuteen. Burnstein kehottaa laatimaan testitapauksia myös suorituskyky- ja kuormitusvaatimuksia vastaan, jotta niihin liittyvät ongelmat havaittaisiin ennen systeemitestausta. White ja Leung (1992) toteavat, että globaalien muuttujien integraatiotestaus voidaan suorittaa vasta, kun kaikki samaa muuttujaa käyttävät komponentit on integroitu yhteen.

2.4.4 Järjestelmätestaus

IEEE:n (Std 610.12-1990) määritelmän mukaan järjestelmätestauksessa arvioidaan valmiin integroidun järjestelmän vastaavuutta määrityksiinsä sekä toiminnallisesti että laadullisesti. Myers et al. (2004, s. 123) nimittää tätä

testausvaihetta korkeamman tason testaukseksi, joka sisältää erillisinä osina toiminnallisuuksien testaamisen sekä järjestelmätestauksen, jossa testataan laadullisia tavoitteita. Myersin lähtökohta korkeamman tason testaukselle on ohjelmistovirheen määritelmä: "Ohjelmistovirhe tapahtuu, kun ohjelma ei toimi loppukäyttäjän järkevästi odottamalla tavalla." Edes täysin kattava yksikkö- ja integraatiotestaus ei näin ollen riitä havaitsemaan kaikkia ohjelmistovirheitä. Määritelmän vuoksi Myers suosittelee (2004, s. 143-144), että järjestelmätestauksessa tulisi olla mukana loppukäyttäjien edustajia sekä ammattitestaajia, joilla ei mielellään ole suoraa kytköstä ohjelmistoa kehittävään organisaatioon.

Burnstein (2003, s. 163-176) listaa toiminnallisuuksien testaamisen lisäksi järjestelmätestauksen osiksi suorituskyky-, kuormitus-, konfiguraatio-, tietoturva- ja toipuvuustestauksen. Myers et al. (2004, s. 130-143) lisää näihin myös käytettävyyden, luotettavuuden, asennuksen, dokumentaation, ylläpidettävyyden sekä tukiprosessien, esimerkiksi varmuuskopioinnin, testaamisen. Myös yhteensopivuus tai muunnettavuus kuuluu testattavien asioiden joukkoon. Näistä esimerkkejä ovat tietojen toimivuus eri ohjelmaversioiden välillä tai uuteen järjestelmään toteutettu mahdollisuus tiedon tuomiseksi vanhoista tietolähteistä. Kaikkia mainittuja osa-alueita ei tarvitse kuitenkaan testata, vaan osioiden valinta tehdään määritysten ja tavoitteiden mukaisesti.

Toiminnallisuuksien testaamisessa on sekä Burnsteinin että Myersin mukaan oleellista testata kaikki määritellyt toiminnot sekä oikeilla että virheellisillä syötteillä. Myers korostaa mitattavuutta korkeamman tason testauksen perustana: se on mahdotonta ilman kirjallisia ja mitattavia tavoitteita. Määrittelyjen lisäksi Myers nostaa järjestelmän dokumentaation testitapausten laatimisen pohjaksi. Burnstein (2003, s. 165) suosittelee systeemitestauksen suunnittelutyökaluksi vaatimusten *jäljitettävyysmatriisin* käyttöä (engl. traceability matrix), jotta testitapausten ja vaatimusten vastaavuuksista voidaan pitää kirjaa. Järjestelmätestauksen suorittavat yleensä muut kuin kehittäjät (Weyuker 1998).

2.4.5 Hyväksymistestaus

Hyväksymistestauksella tarkoitetaan käyttäjän, asiakkaan tai muun valtuutetun tahon suorittamaa ohjelmiston tai komponentin hyväksymiseksi tehtävää testausta, joka suoritetaan aiemmin laadittuja vaatimuksia ja hyväksymiskriteerejä vastaan (IEEE Std 610.12-1990; TTL 2007). Hyväksymistestaus on asiakasprojekteissa usein sopimukseen kirjattu etappi, jolla asiakas hyväksyy tai hylkää järjestelmän toimituksen. Tämän vuoksi hyväksymistestaukseen tulisi valmistautua huolella. Hyväksymistestauksessa havaitut puutteet korjataan, minkä jälkeen testit uusitaan. Vaihtoehtoisesti asiakas voi myös sallia muutoksia alkuperäisiin vaatimuksiin ja hyväksyä testin puutteineen tai eriävine toimintoineen. Jos asiakas on ollut mukana kehitysprosessissa prototyyppien tai väliversioiden avulla, syntyy muutostarpeita yleensä vähemmän. (Burnstein 2003, s. 177)

Burnsteinin (2003, s. 176-178) mukaan testitapaukset tulee laatia yhdessä asiakkaan kanssa vaatimuksiin perustuen ja asiakkaan tulee hyväksyä testitapaukset. Järjestelmätestaukseen laadittuja testitapauksia voidaan käyttää myös hyväksymistestauksessa.

2.5 Regressiotestaus

Regressiotestauksella tarkoitetaan muutosten jälkeen suoritettavaa järjestelmän tai komponentin uudelleentestausta, jolla varmistetaan, etteivät tehdyt muutokset aiheuta tahattomia vaikutuksia ja että järjestelmä toimii muutosten jälkeen

kaikkien määritystensä mukaisesti (IEEE Std 610.12-1990). Leung ja White (1989) pitävät regressiotestauksen perustarkoitusta samana kuin muullakin testaamisella: löytää mahdolliset virheet ja lisätä luottamusta muutetun ohjelman toimivuuteen. Lisäksi regressiotestauksella pyritään säilyttämään ohjelmiston laatu muutosten yhteydessä. Testien uudelleenkäytettävyyden ja laadun vertailun helpottamiseksi versioiden välillä Leung ja White suosittelevat, että regressiotestaus rakennetaan alkuperäisen testisuunnitelman perusteella käyttäen samoja testaustekniikoita ja -mittareita kuin kehitysvaiheessakin,

Regressiotestaus tulisi suorittaa aina, kun ohjelmistoon tehdään pieninkin muutos tai se siirretään uuteen ympäristöön (Kamer et al. 1999, s. 50; Graham et al. 2007). Ylläpitovaiheen kokonaiskustannuksista jopa 50 % ja kaikesta ohjelmiston elinkaareen kuuluvasta testauksesta jopa 80 % voi aiheutua regressiotestauksesta (Harrold & Orso 2008). Kustannusvaikutustensa vuoksi käytettyjen testien määrää tulisi rajoittaa regressiotestausvaiheessa (Harrold & Soffa 1988). Regressiotestaus ei rajoitu pelkästään ohjelmiston ylläpitovaiheeseen, koska myös kehitysprosessin aikana tapahtuvat muutokset tulee testata (Wahl 1999; Harrold 2009). Tämä korostuu erityisesti ketterissä menetelmissä, joissa ohjelmistosta käännetään säännöllisesti uusia versioita. Tällöin regressiotestaus voidaan suorittaa jokaisen keskitetyn käännöksen yhteydessä tai jopa ennen muutosten tallentamista versionhallintaan (Harrold 2009; Harrold & Orso 2008; Chen et al. 2002).

Regressiotestaus voidaan jakaa kahteen vaiheeseen: alustavaan ja kriittiseen. Alustava vaihe käsittää ajan, jolloin ohjelman julkaistuun versioon tehdään muutoksia ja testejä suoritetaan kehitystyön ohessa. Kriittinen vaihe alkaa siitä hetkestä, kun tehtävät muutokset on saatu valmiiksi, minkä jälkeen testaus nousee tärkeimmäksi työvaiheeksi ennen ohjelman uuden version julkaisua. Regressiotestauksen tehostamisen suurimmat kustannus- ja aikatauluhyödyt voidaan saavuttaa juuri kriittisessä vaiheessa, jonka aikataulu voi olla tiukka tavoitellun julkaisupäivän vuoksi. (Rothermel & Harrold 1997)

Leungin ja Whiten (1989) mukaan regressiotestaus voidaan jakaa kahteen tyyppiin sen mukaan, kuinka testauksen kohde on muuttunut: *korjaavaan* (engl. corrective) ja *kehittyvään* (engl. progressive). Mikäli muutokset koskevat vain ohjelmakoodia ja määrittelyt ovat pysyneet ennallaan, pidetään regressiotestausta korjaavana. Jos myös järjestelmän määritykset ovat muuttuneet, regressiotestaus on kehittyvää. Korjaavaa testausta tarvitaan tyypillisimmin havaittujen vikojen korjauksen ja ohjelmiston kehitysvaiheen aikaisten muutosten testauksessa. Kehittyvä testaus kytkeytyy yleensä järjestelmän toiminnallisuuksien, ympäristön tai suorituskykyvaatimusten suurempiin muutoksiin.

Regressiotestauksen kaksi päästrategiaa ovat *täydellinen* sekä *valikoiva* uudelleentestaus (Rothermel & Harrold 1993; Wahl 1999). Täydellisessä uudelleentestauksessa koko järjestelmä testataan alkuperäisillä testeillä muutosten jälkeen. Pienten muutosten kohdalla täydellinen uudelleentestaus ei ole tehokas strategia kustannusten tai tyypillisesti rajallisen projektiaikataulun vuoksi, koska tällöin testataan myös kaikki muuttumattomat toiminnot. Valikoivassa uudelleentestauksessa käytetään vain osaa alkuperäisistä testitapauksista – tarkoituksena on suorittaa vain ne testitapaukset, jotka testaavat ohjelman muuttunutta osaa.

Leung ja White (1989) nostavat regressiotestauksen kahdeksi keskeiseksi ongelmaksi *testien valintaongelman* ja *testisuunnitelman ylläpito-ongelman.* Testien valintaongelmalla tarkoitetaan muutetun ohjelman kattavan testauksen mahdollistavan testijoukon valitsemista olemassa olevista testitapauksista sekä uusien testien tarpeen tunnistamista. Nämä koskevat taulukossa 2.1 listattuja

valikoivan uudelleentestauksen vaiheita 1 ja 3. Testien valintaa käsitellään tarkemmin luvussa 2.5.2.

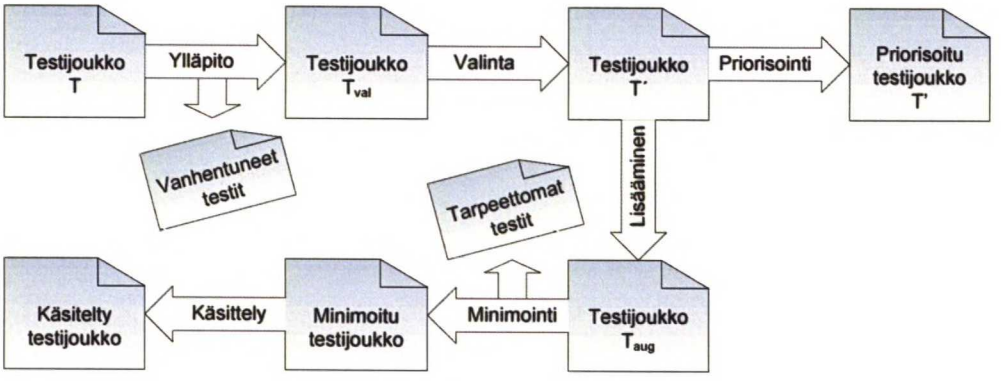
Täydellinen uudelleentestaus ei ole valintaongelman puuttumisesta huolimatta suoraviivainen strategia. Leungin ja Whiten (1989) mukaan pelkkä vanhojen testien uudelleensuorittaminen ei riitä regressiotestaukseksi, sillä testit eivät muutosten jälkeen aina toimi alkuperäisen tarkoituksensa mukaisesti ja ne voivat antaa myös virheellisiä tuloksia. Tämän vuoksi testisuunnitelmaa täytyy ylläpitää jatkuvasti muutosten yhteydessä poistamalla siitä vanhentuneita ja lisäämällä tarvittaessa uusia testitapauksia, jotta testien kattavuus pysyy vähintään alkuperäisellä tasolla. Tätä testisuunnitelman optimointiongelmaa he kutsuvat testisuunnitelman ylläpito-ongelmaksi, joka koskee taulukon 2.1 vaihetta 5. Sen ratkaisu edellyttää testien valintaongelman ratkaisun sekä siihen kuuluvien uusien testien suunnittelun lisäksi myös vanhentuneiden testien siivoamista pois testijoukosta. Leung ja White korostavat testisuunnitelman ylläpidon tärkeyttä myös tiedon siirtämisessä vaihtuvien kehitys- ja ylläpitotiimien välillä. He muistuttavat myös valitun testausjärjestyksen perusteluiden dokumentoinnista. Taulukossa 2.1 mainittuihin testien suoritusvaiheisiin (vaiheet 2 ja 4) liittyy erillisenä ongelmana *testien suoritusongelma*, jolla Rothermel ja Harrold (1996) tarkoittavat sekä testien suoritus- että tulosten analysointivaiheen tehostamista.

1.	Valitaan muokatulle ohjelmalle suoritettavat testit alkuperäisestä testijoukosta.
2.	Suoritetaan valitut testit.
3.	Laaditaan tarvittaessa uudet toiminnalliset ja rakenteelliset testit muokatulle ohjelmalle.
4.	Suoritetaan uudet testit.
5.	Laaditaan muokatulle ohjelmalle uusi testijoukko ja testihistoria edellisten vaiheiden perusteella.

Taulukko 2.1: Valikoivan uudelleentestausprosessin vaiheet (Rothermel & Harrold 1996).

Harrold ja Orso (2008, kuva 1) tiivistävät nämä ongelmat kuvassa 2.4 esitettyihin tyypillisiin regressiotestauksen suoritusta edeltäviin vaiheisiin. Testisuunnitelman ylläpitovaiheet voidaan tunnistaa kuvan nuolista ylläpito ja minimointi. Valintaongelmalla on oma nuolensa. Molemmille ongelmille yhteinen tehtävä on testien lisäämisvaihe. Harrold ja Orso nostavat myös testien priorisointiongelman regressiotestauksen pääkysymysten joukkoon.

Tyypillisessä regressiotestauksen kulussa alkuperäisestä testijoukosta pyritään aluksi siivoamaan muutosten vuoksi vanhentuneet virheelliset testit pois. Tämän jälkeen valitaan suoritettavat testit uudelleentestattavien ja uudelleenkäytettävien testien joukosta – täydellisen uudelleentestauksen tapauksessa molemmat testijoukot suoritetaan kokonaan. Jos resursseista on pulaa, testien suoritusjärjestys tulee priorisoida. Muutosten testaaminen voi vaatia myös uusien rakenteellisten ja toiminnallisten testien lisäämistä testijoukkoon. Jotta regressiotestaus pysyy tehokkaana, tulisi testijoukko minimoida lisäysten jälkeen poistamalla joukosta päällekkäisiä samaa asiaa joko toiminnallisesti tai rakenteellisesti testaavia testitapauksia. Viimeisellä käsittelyvaiheella tarkoitetaan olemassa olevien testien avulla tapahtuvaa uusien testitapausten luontia, mitä voidaan pitää testien lisäämisen erikoistapauksena. Eri vaiheita käsitellään tarkemmin seuraavissa alaluvuissa.



Kuva 2.4: Regressiotestijoukon valinnan ja testisuunnitelman ylläpidon tyypillinen työnkulku.

2.5.1 Regressiotestijoukon ylläpito ja minimointi

Regressiotestijoukon ylläpidolla tarkoitetaan vanhentuneiden testien tunnistamista alkuperäisestä testijoukosta sekä virheellisten testien korjaamista uudelleen toimiviksi (Harrold & Orso 2008). Leungin ja Whiten (1989) määritelmän mukaan *vanhentuneisiin* (engl. obsolete) testitapauksiin kuuluvat ne testit, jotka määritysten muuttumisen vuoksi antavat vääriä tuloksia mukaan lukien suorituksen epäonnistuminen. Vanhentuneiksi katsotaan myös oikean tuloksen antavat testit, jotka eivät enää lisää muutosten jälkeisten testien rakenteellista kattavuutta, ja jotka eivät enää kohdistu alkuperäisen suunnitelman mukaiseen tekijään, kuten esimerkiksi tiettyyn raja-arvoon. Näitä testejä voidaan kutsua myös *päällekkäisiksi* (Harrold & Orso 2008).

Nykyiset kääntäjät tunnistavat automaattisesti perustapaukset, joissa testi kutsuu suoraan muuttunutta tai poistunutta rajapintaa kooditasolla ja raportoivat nämä käännösvirheinä, jolloin nämä testit voidaan poistaa tai muokata ne käyttämään uusittua rajapintaa. Määritysten muutosten vuoksi vanhentuneiden testitapausten tunnistamista ei ole juurikaan tutkittu eikä menetelmiä tunnistamisen automatisoimiseksi ole. Regressiotestijoukon ylläpito on nykyisin lähes täysin manuaalinen työvaihe ohjelmistoteollisuudessa. (Harrold & Orso 2008).

Regressiotestijoukon minimoinnilla tarkoitetaan sekä päällekkäisten testien tunnistamista että niiden poistamista testijoukosta. Testijoukon minimoinnin suurin haaste liittyy siihen, että minimointi heikentää testijoukon kykyä löytää virheitä (Harrold & Orso 2008). Toinen ongelma on liittynyt minimointi-perusteisiin, joita on yleensä käytössä vain yksi. Tätä tilannetta parantamaan Hsu & Orso (2009) ovat laatineet menetelmästään MINTS-ohjelmiston, jonka avulla minimoinnissa voidaan huomioida useampi kriteeri samaan aikaan. Tekijät toivovat tämän johtavan muita tekniikoita parempaan virheiden havaitsemis-kykyyn.

2.5.2 Regressiotestien valinta

Regressiotestien valinnalla tarkoitetaan Harroldin ja Orson (2008) käyttämän määritelmän mukaan testijoukon ylläpidon jälkeen tehtävää testien valintaa siten, että ohjelmaan tehdyt muutokset saataisiin testattua mahdollisimman tarkasti käyttäen mahdollisimman pientä osaa alkuperäisistä testitapauksista. Tällä pyritään testauksen nopeuttamiseen ja kustannussäästöihin.

Leung ja White (1989) luokittelivat testitapaukset kolmeen eri ryhmään: *uudelleenkäytettävät* (engl. reusable), *uudelleentestattavat* (engl. retestable) ja *vanhentuneet* (ks. luku 2.5.1). Uudelleenkäytettävät testit suorittavat sekä määrityksien että koodin osalta muuttumattomia kohtia odottaen muuttumattomia tuloksia. Uudelleentestattavien testitapausten ryhmä sisältää testitapaukset, joiden kohteena oleva koodi on muuttunut, mutta määrittelyt ja odotetut tulokset ovat pysyneet ennallaan. Harrold ja Orso (2008) pitävät tätä luokittelua regressiotestien valintamenetelmien pohjana.

Valintamenetelmien eräs luokittelukriteeri on *turvallisuus* (engl. safety). Turvallisilla valintatekniikoilla poimituilla regressiotesteillä havaitaan samat viat kuin täydellisellä uudelleentestauksella – täydellinen uudelleentestaus on siis määritelmän mukaan turvallinen menetelmä. Mikäli valintamenetelmä ei takaa tätä ominaisuutta, se ei ole turvallinen (Rothermel & Harrold 1996). Esimerkiksi satunnainen testien valinta ei ole turvallinen menetelmä, koska siinä testitapausten joukosta voivat jäädä pois juuri muutoskohtaa testaavat tapaukset. Menetelmän turvallisuus voi myös perustua oletuksille, jotka ovat harvoin voimassa käytännön tilanteissa. Turvallisten menetelmien vastakohtana voidaan pitää *minimoivia* menetelmiä, jotka pyrkivät valitsemaan pienimmän mahdollisen määrän juuri muutoskohtaa testaavista testeistä. (Engström et al. 2009).

Rothermel ja Harrold (1996) ovat määritelleet neljä mittaria testien valintamenetelmien vertailua varten. *Sisältävyys* (engl. inclusiveness) kuvaa muutosten kannalta oleellisten testitapausten valintaa regressiotesteihin ja *tarkkuus* (engl. precision) kykyä karsia muutoksen kannalta turhat testitapaukset pois regressiotesteistä. *Tehokkuus* (engl. efficiency) kuvaa valintatekniikan vaatimaa laskennallista monimutkaisuutta sekä sen automatisoinnin mahdollisuutta. *Yleistyvyys* (engl. generality) kuvaa menetelmän soveltuvuutta käytännön tilanteisiin. Turvallinen valintamenetelmä on aina 100 % sisältyvä, eli se valitsee kaikki muutosta testaavat testitapaukset. Ihannemenetelmä olisi sekä täysin sisältyvä että tarkka, jolloin yhtään ylimääräistä testiä ei valittaisi. Rothermel ja Harrold ovat kuitenkin havainneet, että menetelmien tehokkuus laskee, kun muita arvoja halutaan parantaa. Leung ja White (1991) ovat todenneet, että valikoiva regressiotestaaminen tuo kustannushyötyä vain, jos testien valikoimiseen kuluva aika on pienempi kuin ylimääräisten testien suoritukseen ja analysointiin kuluva aika. Tehokkuuden laskiessa menetelmän kustannukset siis nousevat ja voivat tehdä valintamenetelmästä kannattamattoman käytännön sovelluksia ajatellen.

Valintamenetelmät

Regressiotestien valintaa on tutkittu laajasti, ja tutkimus on synnyttänyt lukuisia valintamenetelmiä (Harrold & Orso 2008). Ensimmäisen, tietovirta-analyysiin perustuvan kieliriippumattoman inkrementaalisen valintamenetelmän julkaisivat Harrold ja Soffa vuonna 1988 (Engström et al. 2009). Suurin osa menetelmistä perustuu ohjelman lähdekoodin tutkimiseen ja on suunniteltu tietyntyyppisille ohjelmointikielille (toiminto- eli proseduraalisille tai oliokielille). Osa menetelmistä toimii käännöksen jälkeisellä välikooditasolla .NET ja Java-ympäristöissä. .NET-ympäristössä välikoodiin perustuvat menetelmät mahdollistavat yhden tekniikan käyttämisen, vaikka järjestelmää olisi toteutettu useammalla ohjelmointikielellä (Skoglund & Runeson 2005). Yleensä menetelmät muodostavat kohdeohjelmasta kontrolli- ja/tai tietovirtakaavion ja yhdistävät siihen testitapausten kattavuustiedon. Ohjelmamuutosten vaikutuspisteet havaitaan näistä kaavioista ja suoritukseen valitaan testit, joihin muutoksella voi

olla vaikutusta. Mikäli ohjelmistoon liittyy tietokanta, jonka tilalla on merkitystä testaamisen kannalta, muuttuu testaaminen haasteellisemmaksi, eivätkä kaikki menetelmät huomioi tätä ylimääräistä tilakomponenttia (Willmor & Embury 2005).

Menetelmiä toimintokielille (C)

Valintamenetelmiä laadittiin ensimmäiseksi toimintokielisiä ohjelmia varten. Rothermelin ja Harroldin (1993;1997) laatima kontrollivirtakaavioon pohjautuva turvallinen valintamenetelmä toimii sekä yksikkö- että integraatiotason testien valintaan ja tunnetaan myös nimellä DejaVu. Se on Engstöm et al. (2009) vertaamista turvallisista menetelmistä tarkin, mutta sen tehokkuus ei ole kaikissa tutkimuksissa voittanut täydellistä uudelleentestausta. Pääsääntöisesti menetelmä näyttäisi tuottavan säästöjä testausaikaan.

Chen et al. (1994) laativat C-kielen rakenteiden (esim. funktiot, muuttujat, makrot) välisiin staattisiin ja dynaamisiin riippuvuuksiin perustuvan turvallisen TestTube-järjestelmän, joka toimii myös systeemitestitasolla. Tekijöiden mukaan menetelmä skaalautuu myös isoihin järjestelmiin, ja sen analyysiaika on Engström et al. (2009) vertaamista turvallisista menetelmistä pienin. Elbaum et al. (2003) muokkasivat menetelmää jättämällä ns. ydinfunktiot, joita suoritetaan vähintään 90 % testitapauksissa, analyysin ulkopuolelle. Tämä tehostaa menetelmää (Engström et al. 2009), mutta samalla luovutaan täydestä turvallisuudesta.

Gupta et al. (1992) esittivät tietovirran määrittely-käyttöparien edestakaiseen läpikäyntiin perustuvan viipalointitekniikan. Menetelmän avulla voidaan joko luoda muutoksia testaavat uudet testit tai vaihtoehtoisesti poimia regressiotestit testijoukosta, mikäli niihin on kiinnitetty tieto suoritetuista määrittely-käyttöpareista. Menetelmä ei suorita koko ohjelman tietovirran uudelleenlaskentaa ja se on havaittu varsin tehokkaaksi. Algoritmi on myös omassa ryhmässään varsin hyvä havaitsemaan vikoja (Barahdi & Mansour 1997; Engström et al. 2009). Ominaisuuksiltaan samankaltaisena viipalointimenetelmänä voidaan pitää myös Agrawal et al. (1993) laatimaa inkrementaalista mallia.

Integraatiotestauksen tasolla tarvittavan regressiotestauksen määrää voidaan rajata Leungin ja Whiten (1990) *palomuurimallilla* (engl. firewall). Regressiotestaus voidaan kontrolli- ja tietovirtariippuvuuksien avulla rajata niihin komponentteihin, joita muutokset koskevat, ja palomuurin ulkopuolelle jäävät komponentit voidaan jättää testaamatta. Pääsääntöisesti testattavaksi valikoituvat suoraan muuttuneiden komponenttien lisäksi ne komponentit, joita muuttuneet komponentit kutsuvat itse sekä ne, jotka kutsuvat muuttuneita komponentteja. Globaalit muuttujat voidaan huomioida palomuurimallin laajennuksella rajaamalla samaa globaalia muuttujaa käyttävät sekä niiden välissä olevat mahdolliset komponentit palomuurin sisään (White & Leung 1992). Menetelmää voi muokata komponenttitasoa tarkemmaksi (Baradhi & Mansour 1997). Palomuurimalli valitsee yleensä paljon testejä ja on sisältyvyyden, tarkkuuden ja tehokkuuden suhteen keskinkertainen menetelmä, mutta se toimii myös integraatiotasolla, mikä erottaa sen muista menetelmistä. (Mansour et al. 2001; Engström et al. 2009)

Volokos ja Frankl (1997) laativat lähdekoodin tekstieroja vertaavan turvallisen menetelmän, joka perustuu siistittyjen C-kielisten lähdekoodien yksinkertaiseen vertailuun UNIXin diff-komennolla yhdistettynä testien suorittamiin ohjelmalohkoihin. Tekijöiden mukaan menetelmä olisi laajennettavissa muillekin kielille. Menetelmän on kuitenkin vain hieman täydellistä uudelleentestausta tehokkaampi (Engström et al. 2009).

Hartmannin ja Robsonin (1990) esittämä minimoiva valintamalli C-kielelle on erittäin tehokas menetelmä kustannuksia ajatellen, mutta samalla testijoukon kyky havaita vikoja heikkenee merkittävästi (Engström et al. 2009). Samaan teoriapohjaan perustuen Mansour ja El-Fakih (1997) laativat simuloidun jäähdytyksen sekä geneettisen algoritmin, jotka ovat lähellä minimoivia. Näiden analysointi on raskasta (Baradhi & Mansour 1997), mutta tekijöiden mukaan niiden pitäisi suoriutua muita menetelmiä paremmin ohjelman koon kasvaessa.

Menetelmiä oliokielille

Rothermelin ja Harroldin DejaVu-menetelmää on muokattu ainakin seuraaville oliokielille soveltuvaksi: C++ (Rothermel et al. 2000), Java-tavukoodi (Harrold et al. 2001) ja .NET/CIL-välikieli (Koju et al. 2003). Kaikki edellä mainitut ovat turvallisia menetelmiä ja ovat ominaisuuksiltaan samankaltaisia alkuperäisen DejaVun kanssa (Engström et al. 2009; Orso et al. 2004). Willmor & Embury (2005) laajensivat Rothermelin ja Harroldin DejaVu-menetelmää tietokanta-sovelluksiin ja pelkistivät siitä vain tietokannan tilan kannalta turvallisen menetelmän, joka pienentää testijoukkoa tehokkaasti.

Kung et al. (1993) muunsivat palomuurimallin soveltumaan olio-ohjelmien tarpeisiin. Tätä mallia kutsutaan *luokkapalomuurimalliksi*, joka muodostetaan analysoimalla valmista ohjelmakoodia laatimalla siitä oliosuhdekaavio. Menetelmän avulla on tarkoitus löytää testausjärjestys, jolla voidaan minimoida tarvittavien tynkien määrä integraatiotestausvaiheessa. Ideana on löytää ensin itsenäiset luokat, jotka yksikkötestataan ja käytetään sen jälkeen integraatiotestauksessa tynkien sijaan. Samaan tapaan testataan kantaluokat ennen periytyviä luokkia, jolloin osa testitapauksista voidaan käyttää uudelleen. Labiche et al. (2000) parantelivat menetelmää ja mahdollistivat sen käyttämisen jo ennen koodin olemassaoloa hyödyntäen normaaleja suunnittelun aikaisia luokkamalleja, esimerkiksi UML:ää (ks. luku 2.4.3). White et al. (2005) ovat myös laajentaneet luokkapalomuurimallia, ja heidän menetelmiensä avulla on saatu suuren yrityksen testausprosessia merkittävästi parannettua. Menetelmän käyttö on ohjannut myös tehokkaasti uusien testien tarpeen tunnistamista. (White et al. 2008).

Skoglund ja Runeson (2005) eivät pidä Kungin alkuperäistä luokkapalo-muurimallia kustannustehokkaana suurten järjestelmien kohdalla. He laativat itse muutospohjaisen menetelmän, jolla testitapaukset valitaan pelkän kattavuustiedon perusteella jättäen luokkien väliset riippuvuudet huomioimatta. Valituksi tulevat ne tapaukset, jotka suorittavat muuttuneita luokkia. Muuttuneet luokat voitiin Java-ohjelmassa analysoida käännettyjen luokkien tiivisteen (engl. hash) avulla. Menetelmän vikojen havaitsemiskykyä ei ole arvioitu.

Orso et al. (2004) yhdistivät Java-ohjelmien analysointiin kontrollivirta- ja palomuuritekniikat, mikä näyttäisi kustannustehokkaalta ratkaisulta myös suurien järjestelmien analysoinnissa, tosin vain tekijöiden oman empiirisen kokeen perusteella. Wu et al. (1999) laativat olio-ohjelmiin soveltuvan funktioiden välisiin riippuvuuksiin perustuvan menetelmän, joka on tekijöiden mukaan tehokas, tarkka ja turvallinen. Engström et al. (2009) eivät ole merkinneet menetelmää turvalliseksi ilmeisesti vähäisestä tutkimuksesta johtuen.

Menetelmiä komponenttiohjelmistoille ilman lähdekoodia

Orso et al. (2001) laativat komponenttiohjelmistoille valintamallin, jonka edellytyksenä oli komponentteihin liitetty tarkka metatieto muutoksista ja komponentin kontrollivirrasta. Menetelmä on sinänsä tehokas, mutta tarvittavan

metatiedon tarkkuus tekee siitä epärealistisen ulkoisten komponenttien suhteen (Mao & Lu 2005).

Sajeev ja Wibowo (2003) laativat oman komponenttiohjelmien valintamallin, joka perustuu komponenttien versiohistoriaan (versio ja muutetut metodit) sekä testien suorittamille metodikutsuille. Mallissa valitaan kaikki testitapaukset, jotka kutsuvat joko suoraan tai epäsuorasti muuttuneita metodeja. Menetelmän toteutus on helppo, mutta se ei ole tarkka (Mao & Lu 2005).

Mao ja Lu (2005) laativat oman mallinsa edellyttäen komponentin toimittajalta erillisen kuvauksen muutosten vaikutuksista rajapintamuuttujiin liitettyjen ehtojen avulla. Malli ei ole yhtä tarkka kuin Orso et al. (2001), mutta selvästi tarkempi kuin Sajeev & Wibowo (2003). Tämänkin mallin vaatima muutostieto ulkopuolisilta komponenttitoimittajilta kuulostaa epärealistiselta, mutta voisi toimia yrityksen sisäisessä komponenttikehityksessä.

Zheng et al. (2007a) laativat I-BACCI -nimisen menetelmän, jolla pyritään suorittamaan regressiotestien valinta binäärikoodin muutoksiin perustuvalla palomuurianalyysillä. Tutkimuksessa selvitettiin myös menetelmän tarvitseman *takaisinmallinnuksen* (engl. reverse engineering) laillisuutta, mikä voi osoittautua ongelmalliseksi, jos lisenssi tämän suoraan kieltää – tosin tämäntyyppisessä käytössä laki on tekijöiden mukaan ainakin Yhdysvalloissa epäselvä. Menetelmä on myös automatisoitu työnimellä Pallino (Zheng et al. 2007b). Toistaiseksi menetelmää ei ole sovellettu Windows-ympäristössä kuin C ja C++ -kielisiin toteutuksiin.

Pasala et al. (2008) ovat myös tutkineet Windows-ympäristön binäärikomponenttien perusteella tapahtuvaa regressiotestien valintaa. InARTS-menetelmä sopii kaikille .NET-pohjaisille ohjelmille, ja huomioi ohjelmiston ajonaikaisen käyttäytymisen toisin kuin I-BACCI. Menetelmä käyttää takaisinmallinnusta kääntämällä ohjelmat takaisin välikielelle, eikä se toimi sekoitetun (engl. obfuscated) koodin kanssa. Tämä sekä mahdolliset lakitekniset seikat rajoittavat sen käyttöä kaupallisten komponenttien kanssa.

Muuhun kuin koodiin perustuvat menetelmät

Regressiotestauksen työmäärä on yleensä arvioitava kustannusvaikutusten selvittämistä varten jo alkuvaiheissa, jotta muutosprojekti saa hyväksynnän. Koodiin perustuvien rakenteellisten valintamenetelmien vaihtoehtona onkin esitetty käytettäväksi ohjelmiston määrittelyihin ja suunnitelmiin pohjautuvia menetelmiä, joita voidaan käyttää jo arviointivaiheen tukena. Nämä menetelmät ovat myös kieliriippumattomia, koska ne eivät tukeudu ohjelman lähdekoodiin, minkä vuoksi myös koodin analysointivaiheet jäävät pois. Lisäeduksi on mainittu korkean tason dokumentaation ja testitapausten välisen jäljitettävyyden käytännöllisempi ylläpito. (Briand et al. 2009)

Chen et al. (2002) ovat laatineet menetelmän, jossa regressiotestien valinnassa käytetään kahta rinnakkaista tapaa: määrittelyvaiheissa laadittuja UML-aktiviteettikaavioita sekä riskianalyysiä. Aktiviteettikaavion perusteella valittuja testejä kutsutaan *kohdistetuiksi testeiksi* (engl. targeted tests). Menetelmä muistuttaa rakenteellisia kontrollivirtaan perustuvia menetelmiä. Testitapauksiin ja ohjelmakoodiin yhdistetään tieto siitä, mitä reittiä ne kulkevat aktiviteettikaavion läpi. Regressiotestit voidaan valita kaaviosta etsimällä ne testitapaukset, jotka kulkevat muuttuneen koodin tai määritysten vaikuttamia polkuja pitkin. Tämän lisäksi valitaan joukko *turvatestejä* (engl. safety tests) riskianalyysin perusteella varmistamaan käyttäjille tärkeimpien toimintojen toimivuus muutosten

jälkeen. Menetelmä ohjaa valitsemaan testitapauksia, joiden merkitys järjestelmän toimivuudelle on suurin. Yksittäisten testitapausten lisäksi menetelmä on sovellettavissa myös käyttötapauksiin. Tekijät pitävät menetelmää asiakaslähtöisenä tapana testien priorisoimiseksi käytettävien resurssien puitteissa.

Briand et al. (2009) esittävät menetelmän regressiotestien valinnalle sekä ylläpidolle perustuen käyttötapaus-, luokka- ja sekvenssikaavioille. Gorthi et al. (2008) esittävät ohjelmiston määritysten perusteella tiettyä esitystapaa noudattaen laadituille aktiviteettikaavioille perustuvan valintamenetelmän, johon voidaan liittää myös riskiperusteista tietoa testien priorisointia varten. Ali et al. (2007) malli muodostaa UML 2.0:n luokka- ja sekvenssikaavioiden avulla laajennetun rinnakkaisen tietovirtakaavion, jonka muutoksiin perustuen suoritetaan testien valinta. Malli huomioi myös dynaamiset sidonnat ja vaikuttaa tarkemmalta kuin muut UML-pohjaiset menetelmät tekijöiden oman tapaustutkimuksen perusteella.

Tsai et al. (2001) esittävät erilaista testiskenaarioihin pohjautuvaa menetelmää, joka hyödyntää valinnassaan viipalointitekniikkaa. Testiskenaariolla tarkoitetaan tässä määritysten ja erillisten testitapausten välisten riippuvuuksien tarkkaa dokumentointia.

Menetelmien arviointia

Engström et al. (2009) ovat vertailleet tehtyjä tutkimuksia mm. edellä mainittujen valintamenetelmien tehokkuudesta. Tulosten ristiriitaisuuksien perusteella he saivat viitteitä, että testin kohteena olevan ohjelman rakenne ja koko vaikuttavat merkittävästi eri menetelmien toimivuuteen, mutta yleisesti ottaen parempi vikojen havaitsemiskyky pudottaa menetelmän tehokkuutta. Lisäksi he havaitsivat, että useimmilla menetelmillä ei saavuteta kustannushyötyjä vaaditun analyysivaiheen raskauden vuoksi, ja jopa tehokkaimmat menetelmät saattavat sopivan kohdeohjelman yhteydessä olla tehottomampia kuin täydellinen uudelleentestaus.

Graves et al. (2001) vertailivat minimoivaa, tietovirtapohjaista sekä turvallista menetelmää satunnaisotannan ja täydellisen uudelleentestauksen kanssa. Turvalliset menetelmät ovat tehokkaimpia virheiden havaitsemisen ollessa pääasia. Satunnaisotanta on kuitenkin hyvä kompromissi puuttuvan analyysivaiheen vuoksi, jos tavoitellaan kustannustehokasta valintamenetelmää. Minimoiville tekniikoille todettiin olevan käyttöä vain silloin, kun testejä ei voida suorittaa kuin hyvin rajattu määrä.

UML:ään perustuvien menetelmien hyvinä puolina voidaan mainita mallinnus- ja suunnitteluvälineiden de facto -standardiaseman lisäksi havainnot siitä, että UML:n käyttö alentaa ylläpitokustannuksia ja parantaa laatua ylläpitovaiheessa monimutkaisten järjestelmien kohdalla, mikäli tekijöillä on UML-osaamista. Sen sijaan yksinkertaisemmissa toteutuksissa UML-kaavioiden laatimiseen ja ylläpitoon kuluva aika voi viedä jopa enemmän aikaa kuin itse muutosten toteutus (Arisholm et al. 2006). Regressiotestien valinnassa menetelmien tarkkuuden arviointi on vaikeaa, mutta se vaikuttaisi olevan heikompi kuin koodiin perustuvilla menetelmillä (Briand et al. 2009).

Regressiotestauksen laajuudesta ja käytännöistä

Graham et al. (2007) mielestä regressiotestaus tulisi suorittaa mahdollisimman laajasti käytettävissä olevien resurssien mukaan. Regressiotestien valinnassa muutoksen vaikutuksia tulisi arvioida yhdessä eri osapuolten (asiakas, käyttäjät, kehittäjät) kanssa, ja sopia, että testausresurssit keskitetään varsinaisten muutos-

kohtien lisäksi niihin osiin, joissa mahdollisia sivuvaikutuksia pidetään todennäköisimpinä tai joiden huolellisempi testaaminen on perusteltua riskinhallinnallisista syistä.

Rosenblum ja Weyuker (1997) ovat laatineet mallin, jonka avulla testien kattavuustiedon perusteella voidaan arvioida regressiotestien valintamenetelmän kustannustehokkuutta. Menetelmä aiheuttaa itsessään analyysikustannuksen, mutta jos regressiotestaus tulee toistumaan säännöllisesti, voidaan menetelmän avulla tehdä perusteltu päätös jonkin valintamenetelmän käyttämisestä tai käyttämättä jättämisestä.

Käytännössä regressiotestien valinta on Harroldin ja Orson (2008) mukaan yleensä manuaalinen työvaihe. Vaatimusten ja testitapausten jäljitettävyysmatriisi on tyypillinen apuväline, koska sitä päivitetään yleensä hyvin. Määritysten muuttuessa valitaan suoritettavaksi alkuperäisiin määrityksiin liittyvät testitapaukset, joita tarvittaessa muokataan vastaamaan muutoksia. Testitapauksia, joihin muutos vaikuttaa epäsuorasti muuttuneen tai uuden määrityksen kautta, voi kuitenkin jäädä valitsematta tällä menetelmällä, minkä vuoksi se ei ole turvallinen. Riskiä voidaan pienentää dokumentoimalla lisätietoa, kuten esimerkiksi testien historiatietoja, valinnan tueksi. Toinen tyypillinen tapa on luottaa testaajien osaamiseen regressiotestijoukon valinnassa, suorittaa valinta satunnaisesti tai yhdistää nämä kaksi menetelmää. Myös täydellinen uudelleentestaus on yleinen menetelmä. Palomuurimallista on raportoitu hyviä kokemuksia ainakin yhdestä suuryrityksestä sekä testien valinnan että lisäämisen suhteen.

2.5.3 Regressiotestijoukon priorisointi

Harrold ja Orso (2008) tarkoittavat regressiotestijoukon priorisoinnilla testitapausten suoritusjärjestyksen laatimista. Järjestäminen valittujen kriteerien mukaan on tarpeen parhaan mahdollisen testaustuloksen saavuttamiseksi, mikäli kaikkia valittuja testejä ei ole mahdollista suorittaa käytössä olevilla testausresursseilla. Priorisoinnin avulla virheet voidaan myös löytää ja raportoida aiemmin, jolloin niiden korjaaminen voidaan aloittaa, eikä kaikkia testejä aina edes tarvitse suorittaa. Harroldin ja Orson mukaan tutkimuksissa on havaittu priorisoinnin olevan tehokas tapa testausprosessin parantamiseen. Elbaum et al. (2002) mukaan järjestämiseen käytetyt kriteerit voivat vaihdella virheiden havaitsemiskyvystä testien kattavuuden optimointiin, ja priorisoinnissa voidaan huomioida myös aiempi testihistoria.

Srikanth et al. (2005) laativat vaatimuksiin perustuvan menetelmän, jossa jokainen uusi vaatimus arvioidaan kolmen muuttujan osalta arvoilla 1-10: vaatimuksen tärkeys asiakkaalle, toteutuksen kompleksisuus ja vaatimuksen epävakaus. Arvoille asetetaan projektikohtaiset painokertoimet ja painotetut arvot summaamalla saadaan jokaiselle vaatimukselle priorisointiarvo. Testitapaus voi testata useampaa vaatimusta, jolloin sen prioriteetti saadaan suhteuttamalla yksittäisten vaatimusten priorisointiarvot testattavien vaatimusten lukumäärään kaikista vaatimuksista. Testausvaiheessa voidaan laskea havaittujen vikojen perusteella arvo myös virheherkkyydelle. Virheherkkyyden huomioiminen priorisoinnissa on perusteltua, sillä Myers et al. (2004, s. 19) sekä Boehm ja Basili (2001) toteavat vikojen keskittyvän tiettyihin ohjelman osiin noudattaen 80/20-sääntöä: 80 % vioista syntyy 20 %:ssa moduuleista. Myös Kim (2003) on havainnut vikojen keskittymisen.

Malleihin pohjautuvia priorisointimenetelmiä on vain muutamia. Ne ovat kustannustehokkaita, mutta voivat olla herkkiä virheelliselle tiedolle (Korel &

Koutsogiannikis 2009). Sapna ja Mohanty (2009a; 2009b) ovat esitelleet UML-aktiviteetti- ja käyttötapauskaavioille perustuvan priorisointimallin, jonka avulla voidaan laatia järjestys laajempien testiskenaarioiden suorittamiseksi. Malliin on yhdistetty myös asiakkaan tekemä toiminnallisuuksien priorisointi ja sitä voidaan käyttää myös kehitysvaiheen testausjärjestyksen optimointiin. UML-kaavioihin perustuu myös Chen et al. (2002) valintatekniikka (ks. luku 2.5.2), jossa priorisointi tapahtuu riskianalyysin perusteella.

Koodipohjaiset priorisointimenetelmät perustuvat kattavuustiedon, ohjelman muutosten sekä testihistorian yhdistämiselle. Menetelmiä ovat esittäneet mm. Rothermel et al. (2001), Jones & Harrold (2003), joiden menetelmä soveltuu myös testijoukon minimointiin, sekä Srivastava ja Thiagarajan (2002), joiden Echelon-työkalu perustuu binäärikoodin vertailuun ja on ollut käytössä Microsoftilla. Wong et al. (1997) yhdistivät ATAC-työkalun avulla testien valinnan, priorisoinnin ja minimoinnin. Heidän mallissaan priorisointi perustui testin kustannusten ja saavutetun kattavuuden maksimoinnille.

Alspaugh et al. (2007) ovat esittäneet testien suoritusajan suhteen priorisoivaa mallia, joka on tehokas tiukkojen aikarajojen puitteissa tapahtuvan testauksen optimoinnille. Hakualgoritmien käyttöä priorisointiin ovat ehdottaneet Li et al. (2007). Niiden etuna on usean kriteerin samanaikainen käyttäminen järjestystä pääteltäessä (Harrold & Orso 2008). Hakumenetelmiä käytetään myös testidatan luomiseen (McMinn 2004).

Priorisointia käytetäänkin ohjelmistoteollisuudessa enemmän kuin valintamenetelmiä, ja etenkin hakualgoritmien käyttö on yleistymässä. (Harrold & Orso 2008)

2.5.4 Regressiotestijoukon lisääminen ja käsittely

Testijoukon lisäämisellä tarkoitetaan muutosten testaamiseksi tarvittavien uusien testien luomista testijoukkoon (Harrold & Orso 2008). Leung ja White (1989) havaitsivat tekemänsä testitapausten luokittelun yhteydessä (katso 2.5.2) tarpeen testien lisäämiselle ja luokittelivat tehtävät lisäykset kahteen eri ryhmään: *uudet toiminnalliset* (engl. new-specification) ja *uudet rakenteelliset* (engl. new-structural). Uudet toiminnalliset testit testaavat muuttuneita määrityksiä sekä niitä toteuttavaa koodia. Uusia rakenteellisia testejä tarvitaan yleensä nostamaan testauksen kattavuus hyväksytylle tasolle muutosten jälkeen.

Lisäämismenetelmät pyrkivät tunnistamaan tarvittavat kriteerit muutosten testaamiselle ja arvioimaan niiden perusteella käytettyä testijoukkoa sekä ohjaamaan uusien testitapausten muodostamista. Menetelmät perustuvat kontrolli- ja/tai tietovirran analysoinnille sekä Kingin (1975) esittämälle symboliselle suorittamiselle. Lisäämisongelmaa ei ole tutkittu niin laajalti kuin valintaa ja priorisointia, eikä nykyisten menetelmien käytännön toimivuutta ole arvioitu kuin pienillä kokeilla. Niitä ei pidetä vielä käyttökelpoisina todellisissa ohjelmisto-projekteissa (Harrold & Orso 2008). Palomuurimallista on kuitenkin raportoitu hyötyjä myös testijoukon lisäämisen ohjaamisessa (White et al. 2008).

Testijoukon manipulointi on lisäämisen erikoistapaus, jossa aiemmin muodostetuista testitapauksista pyritään tuottamaan uusia erilaisia testitapauksia. Esimerkiksi systeemitestien perusteella voidaan luoda yksikkötestejä, jolloin niiden suunnitteluun käytetty työmäärä voidaan hyödyntää muissakin testausvaiheissa. Tämän alueen tutkimus on tuottanut vasta alustavia menetelmiä. (Harrold & Orso 2008)

2.5.5 Regressiotestaus käytännössä

Edellä listattujen regressiotestausmenetelmien käyttö ohjelmistoteollisuudessa on Harroldin ja Orson (2008) mukaan vielä vähäistä. Niiden laajempaa käyttöönottoa estää kolme merkittävää seikkaa. Ensinnäkin tutkimuksissa kehitetyt menetelmät ja työkalut ovat usein raakileita, joiden käyttäminen ja toiminnan automatisointi on vaikeaa. Niitä ei ole sovitettu teollisuudessa yleisesti käytetyille kehitys- tai testaustyökaluille, mikä entisestään vaikeuttaa menetelmien sovittamista kehitysprosesseihin. Toiseksi useimmat rakenteelliset menetelmät vaativat kattavuustiedon tallentamista, mikä ei ole yleinen käytäntö. Kolmantena syynä he mainitsevat, että menetelmiä on arvioitu liian vähän oikeissa suurissa ohjelmistoprojekteissa, minkä vuoksi niiden todellinen tehokkuus ja kustannukset eivät ole ennakkoon arvioitavissa päätöksenteon tueksi, eikä prosessimuutos ole tämän vuoksi perusteltavissa taloudellisesti. Menetelmät eivät myöskään huomioi riittävästi tiettyjä yleisiä tekniikoita, kuten valmiskomponenttien käyttöä ja sovellusten hajauttamista (mm. www-sovelluspalvelut).

Sen sijaan testaustyökalujen ja -ympäristöjen käytännön tilanne on Harroldin ja Orson (2008) mukaan melko kehittynyt. Syynä tähän pidetään työkalujen hyvää saatavuutta ja laajennettavuutta, jota ovat edistäneet avoimet kehitysprojektit, kuten IBM:n käynnistämä Eclipse-projekti[2]. Microsoft-ympäristöjen puolella kehitystyökalujen laajennuksia kehitetään esimerkiksi CodePlex-yhteisössä[3]. Yleisesti käytetyistä testaustyökaluista Harrold ja Orso nostavat esiin xUnit-arkkitehtuuriin pohjautuvat testaustyökalut. Näitä ovat esimerkiksi Java-ohjelmoinnissa käytetty JUnit[4] sekä .NET-ohjelmoinnissa käytetty NUnit[5]. Myös tallennus-toistotyökalut (ks. luku 3.7) ovat levinneet laajalti käyttöliittymätestauksessa, mutta aihetta ei ole juuri tutkittu.

[2] www.eclipse.org
[3] www.codeplex.com
[4] www.junit.org
[5] www.nunit.org

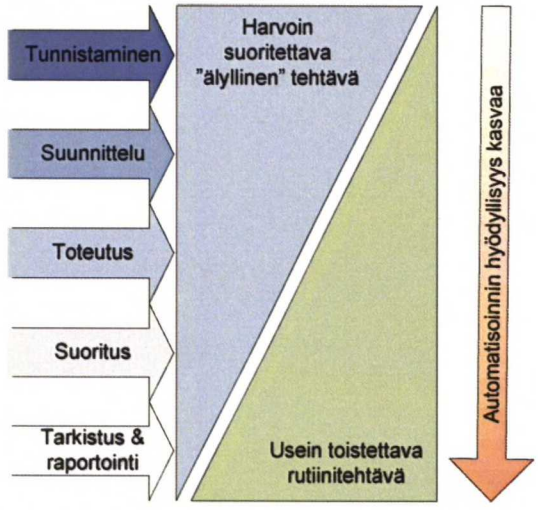
3 Automaattinen testaus

Tässä luvussa käsitellään automaattiseen testaukseen kuuluvia tehtäviä, automatisoinnin hyötyjä, haasteita sekä hyviksi havaittuja toteutusperiaatteita.

3.1 Yleistä

Automaattisella ohjelmistotestauksella ei tarkoiteta pelkkää testien automaattista suorittamista vaan kaikkien testausprosessiin liittyvien tehtävien automatisointia testaustasosta riippumatta. Testauksen automatisoinnilla pyritään yleensä sekä kustannussäästöihin toistuvien työvaiheiden osalta että ohjelmiston laadun parantamiseen kattavammalla testauksella. Hyvin toteutettuna testiautomaatio mahdollistaa molemmat tavoitteet, mutta vaatii tietoista investointia vähintään työn muodossa. (Dustin et al. 1999, s. 3-4; 2009, s. 4-5 & 26-28; Karhu et al. 2009)

Automaation hyödyllisyys korostuu Fewsterin ja Grahamin (1999, s. 17-18) mukaan toistuvissa rutiiniluonteisissa tehtävissä. Kuvassa 3.1 havainnollistetaan heidän esittämäänsä mallia testaustehtävien toistuvuudesta ja siten niiden soveltuvuutta automatisointiin. Tyypillisin automatisoinnin kohde on testien suorittaminen ja niiden raportointi, mutta myös testien suunnittelu- ja toteutusvaihetta voidaan tehostaa esimerkiksi automatisoimalla testiaineiston luomista (Fewster & Graham 1999, s. 19-22).



Kuva 3.1: Testauksen työvaiheet ja niiden suhde automatisointiin

Testien automatisointi on kustannustehokasta, mikäli testien automatisointiin ja automaattisten suoritusten seurantaan käytetty työmäärä on pienempi kuin samojen testien manuaaliseen läpivientiin käytetty aika. Automatisoinnin hyödyt kasvavat testikierrosten määrän kasvaessa, koska automaattisen testin läpiviennissä merkittävin seikka on automatisoinnin vaatima työ, kun taas manuaalisessa testissä eniten aikaa kuluu testin suorittamiseen (Dustin et al. 1999, s. 52). Kaner et al. (2002) huomauttavat kuitenkin, että testikierrosten määrä ilman automaatiota jää käytännössä aina automatisoituja testejä pienemmäksi ja säästöjä tulisi verrata tähän pienempään määrään. Ylimääräisten testikierrosten hyödyksi voidaan laskea myös virheiden aikaisemman löytymisen tuomat säästöt korjauskustannuksissa (Fewster & Graham 1999, 204-206). Berner et al. (2005) huomioivat saman ja toteavat lisäksi, ettei testikierrosten vähäisyys ole juuri

koskaan syynä tehottomaan automatisointiin, vaan väärä automaatiostrategia tai ohjelmiston huono testattavuus. Ramler ja Wolfmaier (2006) ovat kritisoineet suoraviivaisia laskentamalleja ja esittävät itse tapaa optimoida automaattisten ja manuaalisten testien määrä testausbudjetin puitteissa huomioiden testityyppien väliset erot virheriskin pienentämisessä.

3.2 Testiautomaation hyödyt

Fewster ja Grahamin (1999, s.9-10) mukaan testauksen automatisoinnilla voidaan saavuttaa monia etuja verrattuna manuaaliseen testaamiseen. Edellytyksenä tälle on kuitenkin, että testiautomaation kohteet on valittu harkiten ja testausprosessi ja -tekniikat ovat kunnossa jo ennen automatisointia. Tärkeimpänä hyötynä he mainitsevat testien suorittamisen tehostumisen. Tämä on erityisen tärkeää regressiotestauksessa, jonka kustannukset tulisi minimoida (Harrold 2009; Dustin et al. 2009, s. 43). Sama pätee uusien konfiguraatioiden testaamiseen (Dustin et al. 2009, s. 44). Automaation avulla testien määrää voidaan usein myös kasvattaa manuaaliseen testaukseen verrattuna (Graham et al. 2007).

Fewsterin ja Grahamin (1999, s.9) mukaan testien suorituksen tehostuminen mahdollistaa testikierrosten määrän lisäämisen myös kehitysprosessin aikana. Yksikkötestien suorittaminen voidaan ajastaa jokaöiseksi toimenpiteeksi versionhallintaan talletetulle koodille (Harrold 2009; Dustin et al. 2009, s.45-46). Automatisoidut testit voidaan toistaa samalla tavoin kerrasta toiseen ja eliminoida inhimilliset erehdykset testauksessa. Automatisointi vapauttaa näin testaajien aikaa rutiinityöstä niihin testeihin, joiden suoritusta ei voi tai ei kannata automatisoida. Samalla aikaa jää enemmän testaajan työn oleellisimpaan osuuteen: hyvien testien laatimiseen. Myös Patton (2000, s. 220-221) pitää näitä tekijöitä automaattisen testauksen oleellisimpina hyötyinä. Kaikki tämä lisää luottamusta ohjelmiston toimivuuteen ja parhaimmillaan lyhentää myös testaukseen tarvittavaa kalenteriaikaa.

Osa testeistä on käytännössä pakko automatisoida. Näihin kuuluvat esimerkiksi kuormitustestit, joissa järjestelmää kuormitetaan samanaikaisesti jopa tuhansista lähteistä. Myös laajan verkkopalvelun sisältämien linkkien toimivuuden läpikäynti on manuaalisesti pitkäkestoinen, ellei peräti mahdoton toimenpide, mutta koneellisesti tällainen tarkastus on suoritettu nopeasti. (Dustin et al. 2009, s. 45-47)

Testaamisen automatisointi voi myös johtaa vaatimusten ja suunnitelmien tarkempaan dokumentointiin sekä näiden linkittämiseen aina ohjelman toteutustasolle asti, mikä selkeyttää ylläpitoa ja on vaatimus useimpien regressiotestien valintamenetelmien (katso luku 2.5.2) käyttöönotolle (Dustin et al. 2009, s. 46 & 48-49).

3.3 Testiautomaation haasteista

Fewster ja Graham (1999, s. 10-12) toteavat, että testauksen automatisoinnilta ja testaustyökaluilta odotetaan usein täysin epärealistisesti ratkaisua kaikkiin testaukseen liittyviin ongelmiin. Berner et al. (2005) mukaan osasyynä tähän ovat helpolta näyttävät tuote-esittelyt graafisten käyttöliittymien *tallennus-toisto-työkaluilla* (katso luku 3.7 Käyttöliittymätestaus). Ennen automatisointia tulisi aina varmistua, että testausmenetelmät ja testitapaukset ovat sellaisella tasolla, jota kannattaa automatisoida – huonojen testien suorittaminen aiempaa nopeammin ja useammin ei ole hyödyllistä ja saattaa vain heikentää testauksen tehokkuutta (Kaner et al. 2002, s. 98-99). Epärealistiset odotukset johtavat usein

yrittämään testiautomaation liian nopeaa tai laajaa käyttöönottoa (Fewster 2001; Fewster & Graham, 1999, s. 412-414/Amland).

Automatisoidut testit eivät auta uusien vikojen löytämisessä. Niiden rooli on toistaa jo aiemmin laadittuja testejä, jotka yleensä suoritetaan testin laatimisen jälkeen ensimmäisen kerran manuaalisesti, ja viat havaitaan jo siinä yhteydessä. Bachin (1999) ja Berner et al. (2005) havaintojen mukaan jopa 80 % virheistä löydetään manuaalisen testauksen kautta, vaikka organisaatio olisi kehittänyt testiautomaatiota pitkään. Tätä tukee myös Karhu et al. (2009) havainto siitä, että testien kattavuuden parantaminen vaatii käytännössä ihmisen laatimia testitapauksia. Automaation pääkohteena tulisikin pitää regressiotestausta, jolla varmistetaan, ettei ohjelmistoon synny muutosten myötä uusia virheitä (Fewster & Graham 1999, s. 11; Dustin et al. 2009, s. 201-206).

Testiohjelmisto ei voi korvata testaajan osaamista ja ihmisen intuitiota etenkään tulosten arvioinnissa (Bach 1999; Patton 2000, s. 239). Esimerkiksi ruudulla näkyviin väreihin tai muuhun ulkoasuun liittyvien seikkojen analysointi koneellisesti on vaikeaa. Sama pätee ääninä annettavien tulosteiden arviointiin. Testi voi myös vaatia fyysisiä toimenpiteitä, kuten erillisten laitteiden kytkemistä tai käyttöä. Ihmisen korvaaminen tällaisissa tilanteissa ei yleensä ole kustannustehokasta. (Bach 1999; Fewster & Graham 1999, s. 23).

Vaikka automaattiset testit eivät löytäisi ohjelmistosta virheitä, ei sen perusteella voi vakuuttua ohjelmiston toimivuudesta. Muutos ohjelmiston toiminnassa voi pahimmillaan aiheuttaa sen, että jokin vanha testitapaus antaa virheellisen positiivisen tuloksen. Tällöin testi päätyy alkuperäisen tarkoituksensa mukaiseen lopputulokseen ja näyttää menneen hyväksytysti läpi, vaikka muuttuneessa ohjelmassa sen olisi pitänyt aiheuttaa virhe, kun testiä ei ole päivitetty vastaamaan muutoksia. Automatisoinnin vuoksi kyseinen vika säilyy ohjelmistossa siihen asti, kunnes joku ulkopuolinen havaitsee vian ohjelmiston toiminnassa, ja vasta tämän jälkeen vanhentunut testi korjataan. Testejä tulee siis ylläpitää jatkuvasti. Jos automatisoinnin kohteet on alun perin valittu väärin, voi testien jatkuva ylläpito syödä kaiken automatisoinnilta odotetun kustannushyödyn. (Fewster & Graham 1999, s. 11; Patton 2000, s. 238-239; Kaner et al. 2002, s. 102-103)

Automaatioon voi liittyä myös teknisiä ongelmia, joihin ei varauduta riittävästi (Dustin 2002, s. 175; Kaner et al. 2002, s. 104-106, Karhu et al. 2009). Testityökalut voivat toimia virheellisesti ja niiden yhteensopivuus testauksen kohteen kanssa voi osoittautua haasteelliseksi (Fewster & Graham, 1999, s.294 & 412-414/Amland). Testaustyökalun käyttö voi olla syynä havaittuun vikaan, minkä vuoksi virheelliset testitapaukset pitäisi pystyä toistamaan myös manuaalisesti (Patton 2000; Dustin 2002, s. 175).

Testitapausten ylläpitoon tarvittava työmäärä arvioidaan usein väärin, etenkin jos käytetään tallennus-toistotyökaluja. Niiden avulla on helppo automatisoida yksittäisiä testejä, mutta ne ovat herkkiä testattavan ohjelman muutoksille (Fewster & Graham, 1999, s.412-414/Amland; Kaner et al. 2002; Dustin 2002, s. 187-189), eikä tallennettuja testiskriptejä ole helppo käyttää uudelleen. Käyttöliittymätestien tulosten vertailun työmäärä arvioidaan helposti liian pieneksi (Fewster 2001). Myös ohjelmiston suuri konfiguroitavuus sekä lukuisat liittymät eri järjestelmiin vaikeuttavat testauksen automatisointia (Karhu et al. 2009).

Testiohjelmien koulutukseen ja tukemiseen ei aina varata riittävästi resursseja, joita monipuolisten ohjelmistojen tehokkaan käytön opettelu edellyttäisi (Dustin

et al. 2009, s. 77; Fewster & Graham, 1999, s.412-414/Amland). Organisaatiot voivat myös laiminlyödä selkeän vastuunjaon testiautomaation osalta, eikä automatisoinnilta vaadita aina samoja käytäntöjä kuin muulta ohjelmistokehitykseltä. Tämä johtaa henkilösidonnaisuuteen ja pahimmillaan automaation hiipumiseen osaamisen kadotessa henkilömuutosten yhteydessä (Bach 1999; Fewster & Graham 1999, s. 12 & luku 11).

3.4 Onnistuneiden automatisointiprojektien tekijöitä

Edellisessä luvussa käsitellyistä haasteista huolimatta automatisointiprojekteista on myös onnistuneita kokemuksia – usein tosin kasvukipujen kautta, esimerkkejä on listattu mm. Fewsterin ja Grahamin (1999) kirjan luvuissa 14-28.

Tärkeimpänä onnistumistekijänä nousee esiin maltillinen pilotointi ja automaation kehittäminen vaiheittain hyödyntäen aina aiempien vaiheiden oppeja lähestymistapojen soveltumisesta omaan toimintaympäristöön (Dustin et al. 1999, s. 541; Fewster 2001; Fewster & Graham 1999, s. 290-292; Kaner et al. 2002, s. 111; Berner et al. 2005). Dustin et al. (2009, s. 69-71) viittasivat vuonna 2007 tehtyyn kyselyyn, jossa nimenomaan aika- ja resurssipula nousivat suurimmiksi syiksi automaatioprojektien epäonnistumiseen. Pienempiin vaiheisiin jaetulla projektilla tätä riskiä voidaan pienentää.

Toiseksi merkittäväksi tekijäksi on nostettu työkalujen valinta ja testaaminen kohdeympäristössä (Bach 1999; Fewster & Graham 1999, luku 10). Itse laadituissa testiohjelmissa tulee huomioida, että niiden kehitystyö pitää tapahtua samoja periaatteita noudattaen kuin muunkin ohjelmistokehityksen – mukaan lukien näiden työkalujen testaaminen ja uudelleenkäytettävyyden huomioiminen (Patton 2000, s. 14; Fewster 2001; Berner et al. 2005).

Harjoitteluajan järjestäminen on tärkeää automatisointiprojektin onnistumiselle (Fewster & Graham, 1999, s.412-414/Amland; Dustin 2002). Ohjelmistoille olisi hyvä olla organisaation laajuinen tuki, jottei jokaiseen projektiin valita erillisiä työkaluja. Tällöin henkilöstön vaihtuminen projektissa ei aiheuta työkaluosaamisen katoamista. Projektitasolla automatisoinnilla tulisi olla selkeä vastuuhenkilö sekä työkalujen että organisoinnin suhteen (Fewster & Graham 1999, luku 11). Automatisoinnin dokumentaation päivittämisen merkitys korostuu ylläpitovaiheeseen siirryttäessä, jottei projektin aikana kertynyt osaaminen katoa siirron aikana, sillä regressiotestauksen tehokkuudella on suuri vaikutus ylläpidon työmääriin (Bach 1999).

Bach (1999) tiivistää onnistuneen automaatioprojektin salaisuuden seuraavasti: *"Ajattele ensin testausta, vasta sitten automaatiota."* Hän pitää ohjelmiston laatua päämääränä, johon pyritään testauksen avulla. Automaatio on vain eräs keino, jolla testausta voidaan parantaa. Tämä sanelee myös asioiden tärkeysjärjestyksen ja korostaa automaattisen testauksen kohteiden sekä käytettävien menetelmien valinnan tärkeyttä. Oleellisinta on Bachin mielestä katsoa, että perustestit ovat hyvällä tasolla, jonka jälkeen ne voidaan automatisoida. Tämän jälkeen testaajat pääsevät laatimaan ja suorittamaan monimutkaisempia testejä, joiden automatisointi ei ole välttämättä enää kannattavaa. Bach korostaa myös hyvien testiraporttien kehittämisen tärkeyttä, koska niillä säästetään helposti testien analysointiin tarvittavaa aikaa. Hän ei myöskään toisi automaatiota projektiin ennen kuin testauksen kohde on vakiintunut kehityksen alkuvaiheen ongelmista.

Berner et al. (2005) pitävät kohdeohjelmiston testattavuutta tärkeimpänä kustannustekijänä automatisoinnille. Testattavuutta ei yleensä sisällytetä riittävän

selkeästi ohjelmiston vaatimuksiin, ja kun asia havaitaan, sen lisääminen voi olla jo lähes mahdotonta, mikä aiheuttaa automaattisten testien laatimisessa paljon ylimääräistä työtä. Myös Kaner et al. (2002, s. 119-120 & 122-123) pitää testattavuutta tärkeämpänä kuin automatisointia. Kaner et al. lisäksi myös Martin (2005) suosittelee testien ajamista pääosin ohjelmarajapintojen kautta ja tarvittaessa lisäämään testimetodeja ja -palveluita ohjelmistoon. Käyttöliittymien testaamisen helpottamiseksi niissä ei tulisi olla suoraa liiketoimintalogiikkaa mukana, vaan esimerkiksi erilaiset monimutkaiset tarkastussäännöt tulisi erottaa erillisiksi komponenteiksi. Sekä Martin että Berner et al. korostavat yksikkötestauksen kannalta tärkeänä seikkana komponenttien välisten riippuvuuksien pienentämistä. Kaner et al. (2002, s. 120-121) pitää mahdollisena myös yksikkötestauksen automatisointia ilman täydellistä riippumattomuutta noudattaen Myersin inkrementaalista mallia (ks. luku 2.4.3).

Liiketoimintalogiikan testausta tallennettujen käyttöliittymätestien kautta ei pidetä hyvänä tapana. Tallennus-toistotestaus on käytännöllisimmillään järjestelmätestauksen tasolla. Testaus ja testien automatisointi kohdistuu usein juuri tälle tasolle, vaikka yksikkö- ja integraatiotestauksen parantaminen ja automatisointi olisi kustannustehokkaampaa mahdollistaen vikojen aikaisemman havaitsemisen jokaöisten käännösten yhteydessä. Alemman tason testitapauksia voidaan usein hyödyntää myös ylemmillä tasoilla (Berner et al. 2005; Dustin et al. 2009, s. 82-83). Meszaros (2003) tosin puolustaa esimerkkien kera tallennus-toistotestien käyttöä regressiotestauksessa silloin, kun testauksen kohdetta ei ole alun perin suunniteltu testattavuuden kannalta eikä kattavia regressiotestejä ole valmiina. Esimerkkiprojekteissa hän raportoi sekä valmiiden että itse toteutettujen työkalujen käytöstä eri tilanteissa.

Uusien testitapauksien laatimista ja vanhojen ylläpitoa sekä siivoamista tulee jatkaa koko ohjelmiston elinkaaren ajan huolehtien testien dokumentaatiosta ja ylläpidettävyydestä (Bach 1999). Etenkin tallennettujen testien ylläpidettävyyteen on panostettava tietoisesti, jos niistä halutaan uudelleenkäytettäviä ja kestäviä (Dustin et al. 2009, s. 80-81; Berner et al. 2005). Automaattisia testejä tulee ajaa säännöllisesti, sillä pidemmät tauot johtavat helposti siihen, etteivät testit mene enää läpi ja niiden ylläpitoon ei riitä aikaa (Berner et al. 2005).

Jos samassa ympäristössä suoritetaan sekä manuaalista että automaattista testausta, on huolehdittava siitä, etteivät testit käytä samoja lähtötietoja. Suositeltavampaa olisi ajaa automaattitestejä erillisessä ympäristössä (Fewster & Graham, 1999, s.412-414/Amland). Martin (2005) korostaa erillisen vakioidun testitietokannan tärkeyttä.

Testiohjelmistojen kohdalla sekä Fewster ja Graham (1999, s. 129-130) että Dustin et al. (2002, s. 167-169) suosittelevat omien testityökalujen räätälöintiä sekä avoimen lähdekoodin testaustyökalujen hyödyntämismahdollisuuksien arviointia ja muistuttavat, että monesti laajennuksia tarvitaan myös kaupallisiin työkaluihin.

3.5 Testiskriptit automatisoinnissa

Testiskriptillä tarkoitetaan testitapauksen alustamiseen, suorittamiseen ja arvioimiseen tarvittavaa yksityiskohtaista ohjeistusta tai edellä mainitun tiedon sisältävää dokumenttia (IEEE Std 610.12-1990). Testiautomaation tapauksessa testiskriptillä tarkoitetaan myös ohjelmaa, joka suorittaa samat tehtävät ilman ihmisen toimenpiteitä (Fewster & Graham 1999, s. 65-67).

Automaation kannalta hyvien testiskriptien tulisi olla toiminnallisia, eli keskittyä suorittamaan yksi selkeä kokonaisuus, ja ne tulisi toteuttaa siten, että niitä voidaan uudelleenkäyttää eri testitapauksissa. Skriptit pitäisi laatia rakenteeltaan selkeiksi ja ymmärrettäviksi, mikä vaatii huolellista dokumentointia ja kommentointia. Nämä seikat tukevat uudelleenkäytettävyyttä ja helpottavat skriptien ylläpitoa, mikä on tärkeä tekijä onnistuneelle testiautomaatiolle. Yksikään työkalu ei toteuta näitä vaiheita skriptin laatijan puolesta. (Fewster & Graham 1999, s. 68-71).

Testiskriptit voidaan luokitella viiteen ryhmään rakenteensa perusteella (Fewster & Graham 1999, s.75-92): *lineaarisiin*, *strukturoituihin*, *jaettuihin*, *tietopohjaisiin* (engl. data-driven) ja *avainsanapohjaisiin* (engl. keyword-driven). Esimerkiksi käyttöliittymätallennus tuottaa tuloksena lineaarisen skriptin, jossa toiminnot ja mahdolliset vertailut tulevat järjestyksessä peräkkäin. Kyseessä on siis nopea tapa aloittaa automatisointi, mutta näiden skriptien uudelleenkäytettävyys on olematonta. Strukturoitu skripti voi sisältää lisäksi silmukoita tai ehdollisia toimintoja, kuten esimerkiksi vastauksen mahdollisesti esiin nousevaan dialogiin. Tämä mahdollistaa saman skriptin käytön hieman useammassa tilanteessa kuin lineaarisen skriptin tapauksessa, jossa molemmille tilanteille (dialogi tai ei dialogia) tulisi laatia oma skripti.

Jaetut skriptit mahdollistavat uudelleenkäytettävyyden. Tällöin testiohjelman on tuettava toisten skriptien kutsumista. Skriptit voidaan jakaa toiminnallisiin kokonaisuuksiin, ja niiden avulla voidaan huolehtia testeille yhteisten tehtävien suorittamisesta, esimerkiksi kirjautumisesta ja ohjelman perusnavigointi-toimintojen suorittamisesta ilman, että näitä tarvitsee toteuttaa jokaiseen testi-tapaukseen erikseen. Uudelleenkäytettävyys on merkittävä tekijä testiautomaatio-projektien onnistumiselle (Karhu et al. 2009). Skriptien jakamisessa ei kuitenkaan kannata mennä liian pieniin yksiköihin, koska se voi vaikeuttaa skriptien ymmärrettävyyttä (Kaner et al. 2002. s. 113-114).

Tietopohjaiset skriptit suorittavat testejä erilleen tallennetun tiedon perusteella. Tieto voidaan tallettaa esimerkiksi taulukkolaskentaohjelman sivulle siten, että jokaisella sarakkeella on oma merkityksensä, jonka testiohjelmisto ymmärtää. Tieto voi olla vakiomuotoista, jolloin samaa testiä toistetaan vain riveittäin muuttuvilla arvoilla. Vaihtoehtoisesti rivi voi sisältää eri sarakkeissa sijaitsevaa tietoa, joka vaikuttaa suoritettavaan toimintoon. Tietopohjaisella skriptillä voidaan suorittaa suuria testimääriä peräkkäin vain rivejä lisäämällä, mutta käytännössä jokaiseen erilaiseen testiin täytyy laatia oma tietomalli sekä ajuri, joka osaa tulkita tietomallin ja suorittaa sen perusteella halutut toiminnot. Kaner et al. (2002, s. 114) pitävät tietopohjaisia skriptejä helpoimpana tapana toteuttaa monistettavia testejä, joille on saatava paljon erilaisia syötteitä.

Avainsanapohjainen skripti perustuu myös erikseen talletettuun testidataan, mutta nyt keskeisenä ovat avainsanat eli toiminnot, joita halutaan suorittaa. Toiminnon lisäksi talletetaan kyseisen toiminnon tarvitsemat tiedot, jotka ajuriohjelman on tunnistettava. Avainsanapohjaisen testauksen toteuttaminen vaatii paljon perustyötä ajuriohjelmien laatimisessa, mutta toisaalta mahdollistaa testitiimin työskentelyn heille tutuilla ja testattavalle sovellukselle ominaisilla toimintoja kuvaavilla termeillä. Avainsanapohjaisuus tuo myös riippumattomuutta käytetyistä testiohjelmista, koska avainsanojen tulkinta voidaan aina toteuttaa uusiksi ilman, että suurella vaivalla laadittuihin testitapauksiin tarvitsee koskea. Useat lähteet pitävät avainsanapohjaista testiautomaatiota parhaana lähestymis-tapana testauksen automatisointiin varsinkin käyttöliittymätestauksessa (Fewster & Graham 1999, s. 89 & s.450/Buwalda; Kaner et al. 2002, s. 115; Nagle 2002;

Wissink & Amaro 2006; Tang et al. 2008). Mallipohjaisten testaustyökalujen luomia testejä voidaan myös muuntaa avainsanapohjaisiksi testiskripteiksi (Takala et al. 2009).

3.6 Testien suorituksen automatisointi

Testien automaattinen suorittaminen vaatii joko ohjelmallisen testiskriptin tai ajuriohjelman käynnistämistä. Yleensä testit rakennetaan jotain kehysohjelmistoa käyttäen. Nämä sisältävät yleensä mahdollisuuden käynnistää testejä tai testijoukkoja graafisen käyttöliittymän tai komentorivien kautta. Testien suoritus voidaan myös ajastaa tai suorittaa ne aina esimerkiksi ohjelman uuden version kääntämisen jälkeen. Kehitystyökalut, kuten Visual Studio ja Eclipse, voivat sisältää testien suorittamiseen ja automatisointiin tarvittavat perustoiminnallisuudet valmiina. Testiohjelmia saa myös kehitystyökaluun asennettavina lisäosina tai erillisinä työkaluina. Esimerkkejä suosituista testikehyksistä mainittiin luvussa 2.5.5. Käyttöliittymätestauksen erikoispiirteitä käsitellään seuraavassa luvussa. Muiden järjestelmien kanssa kommunikoivien sovellusten testaamisessa Fewster ja Graham (1999, s. 129-130) ovat havainneet usein tarpeelliseksi itse tehtyjen testityökalujen tai laajennusten käytön, jotta kommunikaatiosta saadaan riittävästi tietoa ja jotta rajapintoja voidaan testata tarkemmin.

Fewsterin ja Grahamin (1999, s. 176-190) mukaan testien suorituksen lisäksi myös testien alustus ja siivous kuuluvat varsinaisen automaattisen testaamisen kannalta pakollisiin automatisoinnin kohteisiin, koska jokaisen testikierroksen suorittaminen voi vaatia järjestelmältä samaa lähtötilaa. Valmistelujen ja jälkitoimien automatisointi nopeuttaa testien suorittamista ilman kokonaisautomaatiotakin, koska lähtötilannetta ei tarvitse pystyttää manuaalisesti. Alustustoimet voivat koostua esimerkiksi tietokannan ja hakemistojen tyhjentämisestä tai palauttamisesta varmuuskopiosta, uusien tietojen luomisesta tai vanhojen kopioinneista. Alkutietoja voidaan myös muuntaa muodosta toiseen, mikäli tämä on testien ylläpidon kannalta helpompaa. Siivoustoimet sisältävät lisäksi testin aikaisten tulosten mahdollisen talteenoton ajon jälkeen tehtäviä tarkastuksia varten. Alustus- ja siivoustoimia voidaan suorittaa kokonaiselle testijoukolle, testiryhmille sekä yksittäisille testitapauksille: esimerkiksi jokaisen testitapauksen muuttamat tiedot voidaan palauttaa ennalleen ennen seuraavan ajoa, mutta varsinaisen tietokannan luonti ja lokien tyhjennys suoritetaan vain kerran koko testijoukolle. Mikäli testien suorituksessa havaittiin virheitä, voidaan siivoustoimet jättää ajamatta selvitystyön helpottamiseksi.

3.7 Käyttöliittymätestaus

Graafisen käyttöliittymän (GUI) testaamisessa suurin ero muuhun testaukseen syntyy interaktiivisuudesta. Käyttöliittymässä tehdään toimintoja, esimerkiksi tekstin kirjoittamista, painikkeiden käyttöä, ikkunan koon muuttamista jne. Nämä toiminnot aiheuttavat ohjelmakoodissa tapahtumia (engl. event), joiden perusteella ohjelman suoritus etenee. Käyttöliittymän testaamiseksi testiohjelman tulee kyetä simuloimaan näitä toimintoja käyttäjän puolesta. Tämä voi tapahtua suoraan ohjelmallisella testiskriptillä luokkakirjaston kautta tai *tallennus-toisto-työkalua* hyödyntäen. Jälkimmäiset tallentavat käyttäjän tekemät toiminnot toisto-ohjelman ymmärtämäksi skriptiksi. Yleensä tallennetut skriptit mahdollistavat niiden manuaalisen muokkaamisen, jolloin molemmat mallit on mahdollista yhdistää. (Sun & Jones 2004)

Yksinkertainen selainpohjainen käyttöliittymä aiheuttaa pelkkiä http-kutsuja palvelimelle, jolloin toimintojen toistamiseksi riittää näiden kutsujen toistaminen. Lomakkeella lähetettävät tiedot on mahdollista asettaa kutsuun ilman selaintakin. Selainkäyttöliittymissä on kuitenkin usein laajennuksia, jotka monimutkaistavat tilanteen samankaltaiseksi kuin muissakin graafisissa käyttöliittymissä. Ennen lomakkeen lähetystä selaimessa suoritettava skripti voi tarkastaa tietojen sisällön ja estää lomakkeen lähetyksen palvelimelle sekä näyttää virheviestin, jos tiedoissa on virheitä (Wu & Offutt 2002). Näihin reagoimiseksi testiohjelman on kyettävä kommunikoimaan selaimen kautta, eikä testiä voi ajaa suoraan kutsuna palvelimelle, mikäli halutaan testata käyttöliittymän oikea toiminnallisuus.

Käyttöliittymätestauksen suorittamiseen tarvitaan yleensä laajennuksia alla olevaan testiohjelmistoon. Selainpohjaisten käyttöliittymätestien tekemiseen Visual Studio 2008[6]:sta on olemassa erillinen Test Edition -versio. Erillisenä pakettina on saatavilla esimerkiksi WatiN[7] sekä erillinen tallennustyökalu. Selenium[8] on vastaava Firefox-selaimen laajennus, jolla laadittuja testejä voi ajaa myös muilla selaimilla eri käyttöjärjestelmissä.

Mielenkiintoinen uusi tekniikka käyttöliittymätestauksen automatisointiin on MIT:n Sikuli-projekti, jonka tarkoituksena on luoda graafisiin komponentteihin perustuva testausohjelmisto. Keskeneräisenä se ei vielä ole käyttökelpoinen laajempaan testaukseen, mutta ideana graafisesti näytöltä tunnistettavien osioiden käyttö testiskripteissä tekee niistä hyvin luettavia ja lisäksi riippumattomia ikkunan koosta tai pienistä ulkoasun muutoksista. (Yeh et al. 2009; Chang et al. 2010)

3.8 Testitulosten varmistaminen ja raportointi

Testin suorituksen jälkeen sen tulos on varmistettava vertaamalla saatuja tuloksia odotettuihin tuloksiin. Vertailuja voi joutua tekemään useitakin, sillä jos ruudulla näkyvät tulokset ovat oikein, ei mikään takaa, että ne ovat myös tietokannassa oikein. Vertailujen automatisointi onkin hyödyllisimpiä kohteita automatisoinnin kannalta, paitsi jos vertailut ovat vaikeita suorittaa koneellisesti. Vertailutyökalut eli *vertailijat* (engl. comparator) voivat kuulua testien suoritustyökaluun tai olla erillisiä. (Fewster & Graham 1999, s. 103-104 & 113)

Testijärjestelmän on saatava odotetut tulokset käyttöönsä voidakseen suorittaa vertailut. Tulokset voidaan tuottaa valmiiksi jo testiä suunniteltaessa, kuten testilähtöisessä kehityksessä, jolloin testi voidaan suorittaa automaattisesti alusta alkaen. Testit voidaan suorittaa ensimmäisellä kerralla myös käsin, jonka jälkeen saatu tulos varmistetaan ja siirretään testitapauksen odotetuksi tulokseksi. Tämä voi olla kätevää etenkin, jos tulos on esimerkiksi pitkä XML-dokumentti. Vertailuarvo voidaan hakea myös testioraakkelilta, mikäli sellainen on käytettävissä. (Fewster & Graham 1999, s.102-103)

Vertailut voidaan suorittaa joko dynaamisesti testin suorituksen aikana tai testin suorituksen jälkeen. Dynaamisella vertailulla testiskriptiin saadaan logiikkaa, jolloin se voi reagoida palautteisiin ja virhetilanteisiin, mutta vertailu myös monimutkaistaa skriptejä ja tekee niistä herkempiä muutoksille, mikä lisää testien ylläpitokustannuksia. Testin suorituksen jälkeen tehtävä vertailu voi kohdistua vain talletettuihin lopputuloksiin. Jos skripti tallettaa suorituksen aikana

[6] http://www.microsoft.com/visualstudio/
[7] http://watin.sourceforge.net/
[8] http://seleniumhq.org/

välituloksia jälkivertailuja varten, kyseessä on aktiivinen jälkivertailu. Jos vertailu kohdistuu vain lopputuloksiin, puhutaan passiivisesta jälkivertailusta. (Fewster & Graham 1999, s.107-111)

Tulosten vertailu on yksinkertaisimmillaan suoraa merkkijonojen tai numeeristen arvojen vertailua. Tämä ei aina riitä, koska tuloksissa voi olla muuttuvia arvoja, kuten esimerkiksi aikaleimoja, tunnisteita tai mittaustietoa. Tulosten järjestys voi myös vaihdella. Näiden tilanteiden vertailussa tarvitaan kompleksisia vertailijoita, jotka osaavat suodattaa osan tiedoista vertailun ulkopuolelle ja tarvittaessa tulkita vastauksen oikeellisuutta sen rakenteen perusteella esimerkiksi käyttämällä *säännöllisiä lausekkeita* (engl. regular expressions), joita tuetaan monissa ohjelmointi- ja skriptikielissä. Kompleksiset vertailut ja suodatukset mahdollistavat laajemman automatisoinnin, mutta niiden käyttö lisää virheiden havaitsematta jäämisen riskiä ja tekevät vertailuskriptistä monimutkaisemman. (Fewster & Graham 1999, s. 114-142)

3.9 Testien ja testiaineistojen suunnittelun automatisointi

Testiautomaatio keskittyy yleensä testien suorittamisen ja raportoinnin automatisointiin, mutta myös testien suunnittelu- ja toteutusvaihetta on mahdollista tehostaa automaation avulla. Yksikään työkalu ei tue toistaiseksi koko testausketjun automatisointia (Dustin et al. 2009, s. 74), mutta mallinnukseen pohjautuvat työkalut ovat jo melko kattavia (Blackburn et al. 2004). Kaner et al. (2002, s. 116-117) suosittelevat testidatan luomisen pitämistä erillään testin suorituksesta.

Kuhn et al. (2009) ehdottavat testiaineiston luomista kombinatorisin menetelmin laajemmin kuin pelkkiä muuttujapareja käyttämällä. Useamman muuttujan yhdistelmillä on havaittu olevan parempi kyky havaita virheitä. Tehokkaasti yhdisteltynä testijoukot pysyvät vielä kohtuullisen kokoisina. Tätä varten luotu ACTS-työkalu[9] on saatavilla ilmaiseksi. ACTS ei luo aineistoa automaattisesti, mutta osaa muodostaa testaajan antamien tietojen perusteella erilaiset 2-6 muuttujan yhdistelmät, joita voidaan käyttää syötteenä tietopohjaiselle testijärjestelmälle. XML-aineistojen luomiseksi Bertolino et al. (2007) ovat laatineet TAXI-työkalun[10], joka on ladattavissa myös verkosta.

Ohjelmakoodiin perustuvia testiaineistojen luontimenetelmiä ovat ehdottaneet esim. Korel ja Al-Yami (1998), mutta etenkin hakupohjaiset menetelmät ovat nousseet esiin lupaavimpana tekniikkana (McMinn 2004; Arcuri & Yao 2007). Microsoftin tutkimusyksikkö on laatinut rakenteellisten testitapausten luomiseksi Pex-työkalun[11], joka on vielä kehitysvaiheessa. Pexin tarkoitus on auttaa kehittäjiä laatimaan kattavia yksikkötestejä symbolisen suorittamisen avulla (Barnett et al. 2009). Työkaluun kuuluu nykyisin myös Moles-paketti rajapintojen jäljittelyä varten (engl. mocking). Ohjelmakoodista luodut testit testaavat luonnollisesti vain koodin toteuttamia ominaisuuksia, eivätkä voi havaita mahdollisia toiminnallisia puutteita. Tämän vuoksi testejä pitääkin määritellä erillään toteutuksesta. Koodiin pohjautuvat menetelmät vain tukevat testauksen kattavuutta.

UML:n käyttötapaus-, tila- ja aktiviteettikaavioihin pohjautuvia testien luontimenetelmiä ovat ehdottaneet mm. Hartmann et al. (2000), Offutt et al. (2003),

[9] http://csrc.nist.gov/groups/SNS/acts/index.html
[10] http://www1.isti.cnr.it/ERI/TAXI/taxi_index.html
[11] http://research.microsoft.com/en-us/projects/pex/default.aspx

Nebut et al. (2006), Samuel et al. (2008), ja Sun et al. (2009). Osa menetelmistä soveltuu pelkkään testiaineiston luomiseen, mutta osa pyrkii myös tunnistamaan ja luomaan tarvittavat testit vaatimusten ja mallin perusteella. Näistä Hartmannin malli yhdistää testien luomiseen myös niiden suorittamisen.

Yuan ja Xie (2006) ovat laatineet komponenttiohjelmistojen integraatiotestien luomiseen soveltuvan työkalun, joka perustuu ajonaikaiseen kutsuketjujen rajoitusten analysointiin. Tämän perusteella järjestelmä laatii testattavasta ohjelmasta tilakaavion, minkä avulla tarvittavat testitapaukset voidaan muodostaa. UML-pohjaisiin menetelmiin verrattuna tämänkaltaisia menetelmiä voidaan hyödyntää niissä tilanteissa, joissa mallia ei ole saatavilla tai sitä ei voida purkaa suoritusta analysoimalla.

3.10 Kriteerit automatisoinnille

Automatisoitavien testien järjestystä voi laatia erilaisilla kriteereillä. Fewster ja Graham (1999, s. 230-231) muistuttavat, että pienikin automatisointi voi olla kannattavaa, jos se kohdistuu oikein. Kohteita valitessa kannattaa arvioida testin suorituskertojen ohella myös automatisoinnin vaatimaa työmäärää, joiden avulla voidaan laskea takaisinmaksuaika. Berner et al. (2005) mielestä kymmenen suorituskertaa on yleisesti ottaen testin automatisoinnin kannalta riittävä määrä, jotta automatisoinnin vaatima työ jää pienemmäksi kuin testien suorittaminen manuaalisesti. Myös testien tärkeys ohjelmiston vaatimusten ja toiminnallisuuksien kannalta sekä aiempi virheiden havaitsemiskyky voidaan huomioida päätöksessä (Fewster & Graham 1999, s.549-552/Smale). Toinen näkökulma on keskittyä automatisoinnissa ensisijaisesti niihin testeihin, joita ei voi suorittaa helposti käsin, kuten suorituskyky- ja kuormitustestaus (Kaner et al. 2002, s. 95-96; Berner et al. 2005).

Kehitystä tukeva eli ohjelmiston kehitysversioille usein tehtävä testaus katsotaan useissa lähteiksi tärkeimmäksi automatisoinnin kohteeksi (Kaner et al. 1999, s. 197 & 281; Kaner et al. 2002, s. 95; Dustin 2002, s. 207-209). Tähän ryhmään kuuluvat yksikkötestit, joiden ajo olisi hyvä suorittaa aina käännösten yhteydessä, jotta virheet paljastuisivat heti, sekä savutestit (ks. luku 3.10), joilla varmistetaan ohjelman kriittisten ja helposti testattavien toiminnallisuuksien olevan kunnossa ennen kuin testaus siirtyy varsinaisille testaajille. Savutesteihin voi kuulua yksinkertaisia tallennettuja käyttöliittymätestejä. Kaner et al. (2002, s. 126) ja Berner et al. (2005) suosittavat testien alustus- ja siivoustoimenpiteiden automatisointia kehitystä tukevana toimenpiteenä. Berner et al. lisää myös tulosten raportoinnin sekä testiaineiston luomisen samaan ryhmään.

Regressiotestien automatisointi kannattaa aloittaa mahdollisimman aikaisessa vaiheessa, jos tiedetään, että ohjelmistoa tullaan päivittämään ylläpitovaiheessa usein tai jos ohjelmistoa kehitetään iteratiivisella mallilla. Harvoin, esimerkiksi kerran vuodessa, ajettavia testejä ei kannata automatisoida (Fewster & Graham, 1999, s. 9 & 11; Dustin 2002, s. 201-205).

Käyttöliittymätestauksen automatisointi ei kannata tilanteessa, jossa käyttöliittymän oletetaan muuttuvan merkittävästi ohjelmiston elinkaaren aikana. (Fewster & Graham, 1999, s. 22). Vakiintuneen käyttöliittymän kohdalla Kaner et al. (1999, s. 197 & 281) pitävät käyttöliittymätallennusta toimivana tapana vapauttaa testaajat rutiininomaisten testien suorittamisesta. Heidän mielestään jopa pelkän tallennuksen toistaminen ja silmämääräinen tulosten tarkastaminen on järkevää, jotta vältytään virhepainalluksilta testin yhteydessä. Kokonaisratkaisuna testaamiseen Kaner et al. (2002, s. 103-104 & 106-107) eivät kuitenkaan

tallennus-toistomenetelmää pidä ja he kehottavat aina varautumaan testejä laatiessa käyttöliittymän muutoksiin.

4 Palvelusuuntautunut arkkitehtuuri ja testaus

Tässä luvussa käsitellään palvelusuuntautuneita arkkitehtuureja: mitä niillä tarkoitetaan ja mitä erikoispiirteitä niiden testaamiseen liittyy keskittyen palvelupyyntöihin ja niiden suorittamiseen.

4.1 Palvelusuuntautunut arkkitehtuuri (SOA)

Palvelusuuntautunut arkkitehtuuri, lyhennettynä *SOA*, voidaan määritellä W3C:n (2004b) tapaan yksinkertaisesti "joukoksi komponentteja, joita voidaan kutsua ja joiden rajapintakuvaukset voidaan julkaista ja löytää". Sprott ja Wilkes (2004) pitävät tätä määritelmää liian teknispainotteisena, ja suosittelevat käyttämään CBDI-foorumin käyttämää määritelmää, joka huomioi palveluajatteluun liittyvien käytäntöjen merkityksen paremmin:

"Menettelytavat ja puitteet, jotka mahdollistavat sovellusten toiminnallisuuksien tarjoamisen ja käytön palveluina, jotka on julkaistu käyttäjän kannalta oleellisella tarkkuudella. Palveluita voidaan kutsua, julkaista ja löytää, ja niiden toteutus on piilotettu standardeihin perustuvan yksinkertaisen rajapinnan taakse."

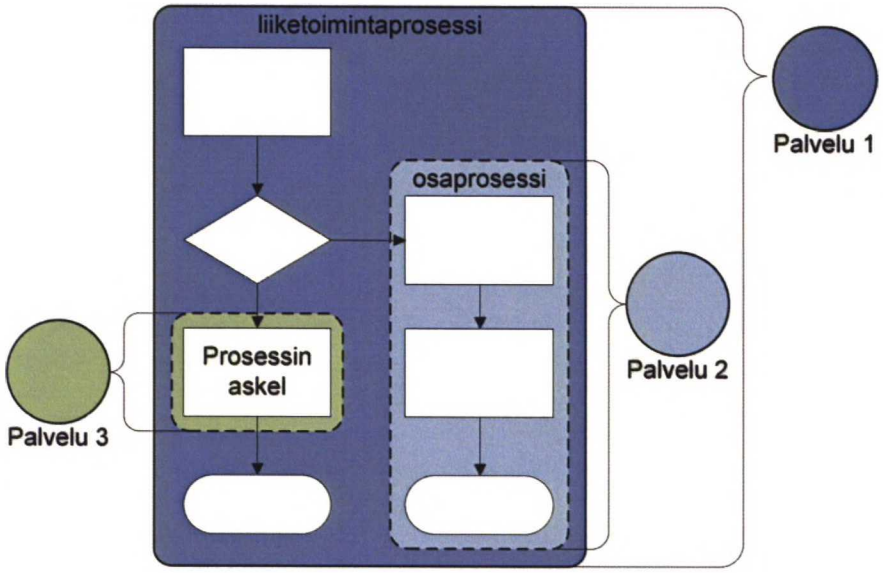
Datz (2004) tiivistää SOA:n pääperiaatteen verkossa kommunikoivien, toisiinsa löyhästi sidottujen liiketoimintaprosesseja suorittavien palvelujen kokoelmaksi. Erlin (2005, luku 3.1) mukaan palvelujen tulee olla toisistaan riippumattomia ja niitä käytetään määritellyn rajapinnan kautta. Näin ollen palveluita voidaan muodostaa myös koostamalla useampi palvelu uudeksi palveluksi oman rajapintansa taakse. Palvelu voi suorittaa prosessiin kuuluvan yksittäisen tehtävän, useammasta tehtävästä koostuvan osaprosessin tai kokonaisen liiketoiminta-prosessin.

Erlin esittämää palvelujen ja prosessien suhdetta on havainnollistettu kuvassa 4.1, jossa esitetty kokonaisprosessi (palvelu 1) voisi olla esimerkiksi tuotteen tilaus, osaprosessina (palvelu 2) laskutus, johon kuuluu tarvittaessa luottokelpoisuuden tarkastus sekä tarvittavat kirjanpitoviennit ja yksittäisenä askeleena (palvelu 3) lähetteen muodostaminen varastolle. Yritys käyttää lähetepalvelua myös sisäisten varastosiirtojen suorittamiseen. Laskutuspalvelua voi käyttää myös laskutettaessa palveluita, joiden tilaus ja hallinta tapahtuvat erillisessä järjestelmässä. Tilaus-palvelua taas käytetään sekä verkkokaupasta että myyjien käyttämästä järjestelmästä.

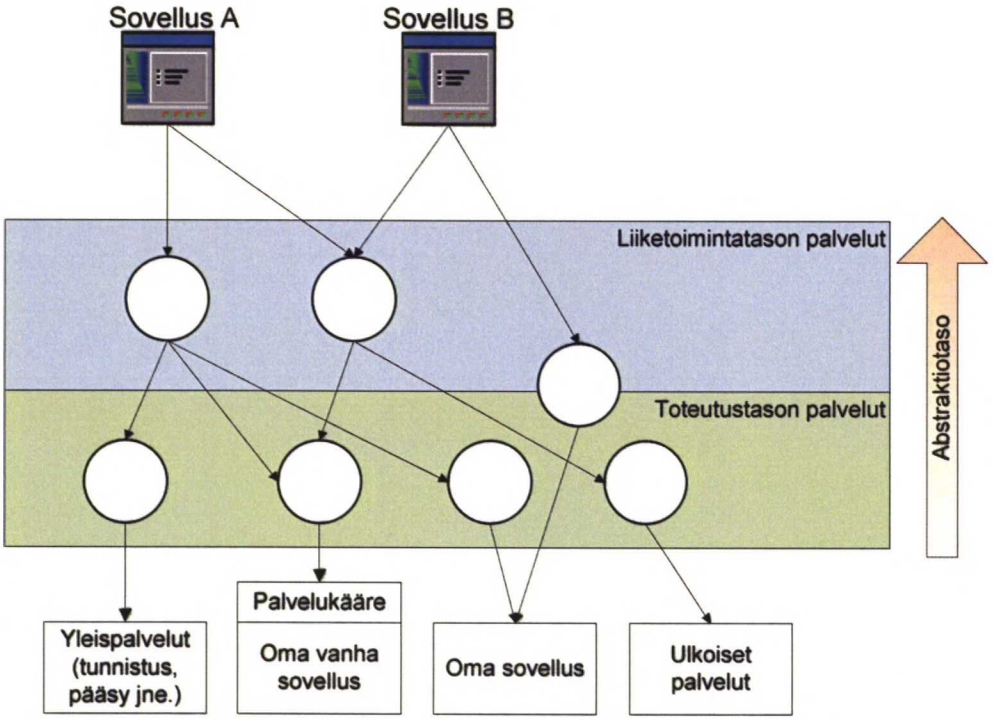
Jotta voidaan puhua palvelusuuntautuneesta ympäristöstä, palvelujen tulee Erlin (2005, luvut 8.3 ja 8.4) mukaan toimia käyttäjänsä kannalta itsenäisinä kokonaisuutena muista palveluista riippumatta (autonomisuus) tarkasti määritellyn rajapinnan läpi (formaali sopimus), joka peittää alla olevan logiikan käyttäjältä (abstraktio) eikä vaadi tilatiedon ylläpitoa (tilattomuus). Näillä periaatteilla palveluista tulee löyhästi sidottuja, mikä mahdollistaa niiden uudelleen-käytettävyyden ja ketjuttamisen. Ketjuttamisella tarkoitetaan sitä, että kutsuttava palvelu voi suorittaa taustalla useita pienempiä palveluita, jotka on julkaistu yleensä alemmalla abstraktiotasolla ja liittyvät lähemmin yksittäisen toiminnon toteutukseen. Tätä on havainnollistettu kuvassa 4.2. Kuvaan on tiivistetty Erlin (2005, kuva 8.3) sekä Sprottin ja Wilkesin (2004, kuvat 1 ja 2) esittämät ydinasiat.

Sprott ja Wilkes (2004) nostavat listalle myös palvelujen merkityksen: palvelu tulisi julkaista aina sellaisella abstraktiotasolla, että se muodostaa käyttäjälleen

merkityksellisen palvelukokonaisuuden. He korostavat myös teknologia-riippumattomuutta rajapintojen osalta. Julkaistuja palveluita voidaan käyttää tarpeen mukaan eri sovelluksista erilaisia toimintoja yhdistellen, kuten kuvassa 4.2 esitetään. Erl (2005, luku 8.3) lisää vielä palvelujen löydettävyyden SOAn peruspiirteisiin, jolloin ympäristö sisältää rekisterin, jonka kautta palveluita voidaan löytää ja uudelleenkäyttää helposti.



Kuva 4.1: Palvelut ja prosessit



Kuva 4.2: Palveluiden tasot ja niiden ketjuttaminen

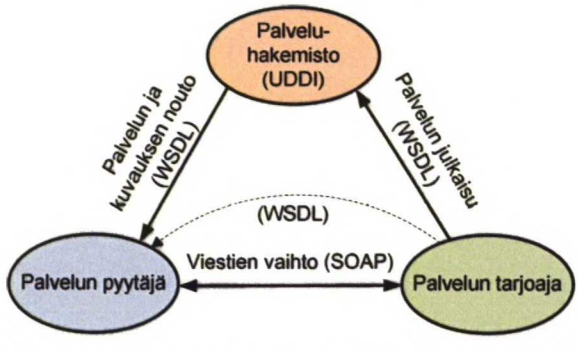
Erlin (2005, luvut 3.4-3.5) mukaan palvelusuuntautunut arkkitehtuuri tuo oikein toteutettuna lukuisia etuja, joista päällimmäisenä on standardeihin nojaava helppo integroitavuus ja yhteensopivuus eri ympäristöjen välillä sekä palveluiden uudelleenkäytettävyys. Palvelumalli voi myös helpottaa vanhojen sovellusten hyödyntämistä muuttuvassa ympäristössä, mikä pidentää niiden elinkaarta. Hyötyjen saavuttamiseksi palveluarkkitehtuuri on kuitenkin rakennettava huolella ja standardoida ympäristö alusta alkaen, jotta se pysyy ylläpidettävänä muutostilanteissa, joihin sopeutumiseksi SOA-ympäristöä yleensä rakennetaan.

SOAn etuna voidaan pitää myös liiketoiminnan ja IT:n yhdistämistä saman ajatusmallin alle. Tätä korostaa Bieberstein et al. (2006) määritelmä, joka toteaa palvelusuuntautuneen arkkitehtuurin olevan "liiketoimintaprosessien ja sitä tukevan IT-infrastruktuurin yhdistämistä turvallisiksi ja standardoiduiksi uudelleenkäytettäviksi komponenteiksi eli palveluiksi, joita voidaan yhdistellä muuttuvien liiketoimintavaatimusten mukaisesti".

4.2 Www-sovelluspalvelut

Www-sovelluspalvelu (engl. web service) on ohjelmistojärjestelmä, joka on suunniteltu tukemaan koneiden välistä vuorovaikutusta tietoverkoissa yhteentoimivien WSDL-kuvauskielellä kuvattujen rajapintojen avulla. Palveluja käytetään kuvauksen mukaisilla SOAP-viesteillä, joissa siirretään yleensä XML-muotoista tietoa HTTP-protokollaa käyttäen. (W3C 2004a)

Palveluita on mahdollista hakea erillisestä rekisteristä, jonka toteuttamisessa suositellaan käytettäväksi OASISin ylläpitämää UDDI-standardia. Palvelu-hakemistojen käyttö ei ole pakollista eivätkä ne ole yleistyneet toisin kuin WSDL-kuvauskielen ja SOAP-protokollan käyttö (Erl 2005, luku 4.1.2), minkä vuoksi rajapintakuvaukset vaihdetaan usein suoraan toimijoiden kesken joko suoraan palvelurajapinnan kautta tai manuaalisesti. Tämä on esitetty www-sovellus-palvelukokonaisuutta esittävään kuvaan 4.3 merkityllä katkoviivalla.



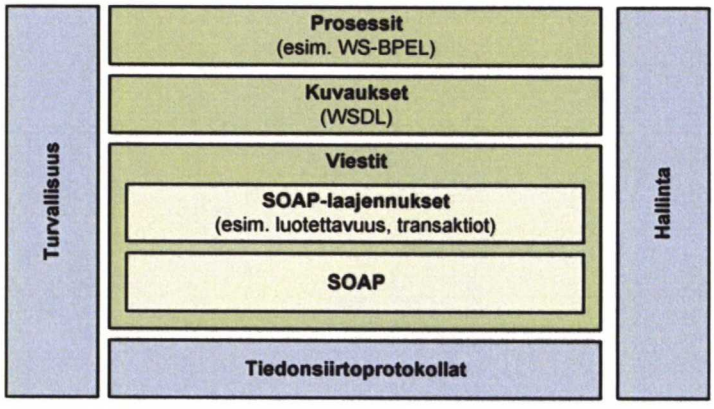
Kuva 4.3: Www-sovelluspalveluiden toimintamalli

Www-sovelluspalvelujen kerrosmalli on esitetty kuvassa 4.4, joka on laadittu W3C:n (2004a) arkkitehtuurikuvauksen perusteella. Kuvan vihreät osat pohjautuvat yleensä XML:n käyttöön kuvauskielenä. Protokollapinon pohjalla on XML-pohjainen viestinvaihtoprotokolla SOAP. Viestien kuljetukseen voidaan käyttää mitä hyvänsä siirtoprotokollaa, toki yleisimmin käytössä on HTTP(S). SOAP-protokollalle voidaan laatia laajennuksia (W3C 2004a). Eräs esimerkki SOAP-laajennuksista on WS-Security, joka standardoi viestien suojaamiseen

liittyviä asioita (OASIS 2006). WSDL-kuvauskielellä määritellään palvelun sisältämät liittymät ja sanomarakenteet.

Protokollapinon päälle on mahdollista laatia näitä viestinvaihto- ja sanomakuvaus-standardeja hyödyntäviä laajennuksia. Esimerkki tällaisesta on WS-BPEL, jolla kuvataan liiketoimintaprosesseja, joiden liittymät kuvataan WSDL:n avulla (OASIS 2007).

Www-sovelluspalveluiden yhteensopivuutta pyritään edistämään WS-I:n (Web Services Interoperability Organization) toimesta. WS-I laatii suosituksia, joita noudattamalla julkaistujen palveluiden pitäisi olla mahdollisimman ympäristö-riippumattomia. WS-I tarjoaa myös testityökaluja sekä niiden toiminnallisia määrityksiä kehittäjien käyttöön. (WS-I 2010).



Kuva 4.4: Www-sovelluspalvelujen arkkitehtuuripino

Www-sovelluspalvelut ja SOA eivät ole sama asia. Tähän väärään käsitykseen voi törmätä varsin usein. SOA-ympäristö on mahdollista toteuttaa muillakin teknologioilla (Datz 2004). Www-sovelluspalvelut ovat kuitenkin helppo ja luonnollinen lähestymistapa SOA-ympäristöä rakennettaessa. Erl (2005, luku 5) pitää nykyaikaista SOA:a lähes riippuvaisena niistä, koska laajennuksineen www-sovelluspalveluprotokollat tarjoavat valmiina monipuolisen SOA-ympäristön tarvitsemat toiminnallisuudet mukaan lukien prosessit, transaktionaalisuus, luotettava viestinvälitys ja tietoturva.

4.3 Palveluiden testaukseen liittyviä erikoispiirteitä

Palveluiden testaamiseen liittyy erikoispiirteitä, jotka korostuvat etenkin silloin, kun testataan ulkoisia palveluita. Tästä huolimatta niiden testaamiseen liittyvä tutkimus on ainakin julkaisujen valossa ollut yllättävän pientä (Bartolini et al. 2009b).

Palveluiden testaaminen muistuttaa kaupallisten valmiskomponenttien testaamista. Kummassakaan tapauksessa käytössä ei ole kuin rajapintakuvaus, joten vain toiminnallinen testaaminen on mahdollista. Bartolini et al. (2009a) ovat tosin ehdottaneet *testattavien palveluiden* mallia, joka mahdollistaa kattavuus-tiedon liittämisen palvelurajapintoihin. Bertolinon (2009) mielestä testaustekniikat perustuvat yleisesti ottaen mahdollisuuteen havainnoida ja kontrolloida testin kohdetta. Kaupallisissa komponenteissa testaamiseen on mahdollista käyttää myös takaisinmallinnukseen tai jäljitykseen liittyviä menetelmiä, mutta ulkoisten palveluiden kohdalla tämä ei onnistu. Palvelujen testaamisessa on turvauduttava

joko oikean palvelurajapinnan käyttöön tai laadittava ulkoisia rajapintoja jäljittelevä sovellus, mikä voi vaatia paljon resursseja. Testaamiseen voi liittyä lisäksi palvelutason varmistaminen esimerkiksi vasteajan tai kokonaiskustannusten osalta (Canfora & Di Penta 2006).

Palveluun tehtävät muutokset eivät välttämättä näy millään tavoin käyttäjälle rajapintakuvauksessa, mutta niillä voi silti olla epäsuoria vaikutuksia palvelun toimintaan. Palveluntarjoajalla ei myöskään ole tietoa siitä, miten heidän palveluaan kutsujapäässä käytetään. Tämä on yhtenäistä perinteisten ohjelmakomponenttien käytön kanssa (Bruno et al. 2005). Weuykerin (1998) mielestä tällaisissa tilanteissa komponentin käyttäjän vastuulla on suorittaa ohjelmistolle mahdollisimman laaja testaus laadun varmistamiseksi, koska komponentin toimittaja ei voi mitenkään taata tai testata oman komponenttinsa toimivuutta kaikissa mahdollisissa tilanteissa.

Www-sovelluspalveluiden testaamisen toinen merkittävä erikoispiirre liittyy niiden dynaamiseen sidontaan (Offutt & Xu 2004). Canforan ja Di Pentan (2006) mukaan kaikki liittymät eivät välttämättä ole edes tiedossa ennen testausta palvelusuuntautuneessa ympäristössä. Samalla testaamisen haasteeksi nousee testien kattavuus: kuinka saada kaikki mahdolliset liittymäyhdistelmät testattua ja varmistettua, ettei mikään yhdistelmä aiheuta toiminnallisia ongelmia esimerkiksi palvelurajapinnan yhteensopivuuden osalta.

Palveluiden regressiotestauksen suurimmiksi ongelmiksi Canfora ja Di Penta (2006) nostavat tiedonsaannin ja testiympäristöjen puuttumisen. Palveluntarjoajalla ei välttämättä ole kaikkien palvelun käyttäjien yhteystietoja, jolloin palvelun muutoksista tiedottaminen esimerkiksi suoraan sähköpostilla ei onnistu. Käyttäjien kannalta tilanne eroaa oleellisesti valmisohjelmien päivittämisestä, koska näiden aikataulu ja testaaminen ovat yleensä käyttäjän omassa hallinnassa.

Ulkoisten palveluiden kohdalla käyttäjille ei välttämättä ole edes tarjolla erillistä testiynpäristöä. Tällöin ongelmaksi voi muodostua uuden liittymän tuotantoympäristössä tapahtuvan testaamisen aiheuttamat kustannukset. Canforan ja Di Pentan (2006) mukaan olisikin suositeltavaa, että palvelun uusi versio julkaistaisiin aina käyttäjien testattavaksi tuotantojärjestelmän ulkopuolella.

Bruno et al. (2005) mukaan erillisen testiympäristön puute tuottaa lisähaasteita myös silloin, kun palvelun suorittaminen aiheuttaa "peruuttamattomia" vaikutuksia, kuten esimerkiksi varauksen teko ulkoiseen järjestelmään. Tällöin testaus tulisi aina suorittaa erillisessä ympäristössä, jossa tapahtumaa ei viedä loppuun saakka tai palvelu tarjoaa erillisen mahdollisuuden simuloituun suoritukseen ilman muutosten tallentamista.

4.3.1 Testimenetelmiä ja -työkaluja

Perinteisiä testiohjelmia, joilla voidaan kutsua palvelua ja tarkastaa saatu vastaus, on saatavilla useita sekä ilmaisina että kaupallisina versioina (ApTest 2010), mutta ne tukevat käytännössä vain yksittäisten palveluiden kutsumista ja vastauksen tarkastamista. soapUI[12] on eräs suosittu työkalu, josta on sekä kaupallinen että ilmainen versio. Ohjelman kotisivun mukaan sitä on ladattu verkosta yli 1,4 miljoonaa kertaa.

[12] http://www.soapui.org

Näiden lisäksi palvelujen testaamisen tehostamiseksi on tutkimuksissa esitetty muutamia erilaisia malleja, jotka koskevat lähinnä testien automaattista suorittamista sekä testiaineistojen luomista.

Bartolini et al. (2009b) ovat laatineet WS-TAXI-nimisen alustavan testikehikon www-sovelluspalvelujen testaamiseen. Se yhdistää heidän aiemmin XML-aineistojen generointiin laaditun TAXI-ohjelmansa sekä ilmaisversiona saatavan soapUI:n. He ihmettelevät, miksei testityökaluissa ole toistaiseksi yhdistetty testiaineiston luomisen ja testien suorittamisen automaatiota, vaikka www-sovelluspalvelujen alusta tarjoaa siihen erinomaiset mahdollisuudet. Eräs syy voi olla WS-TAXI:stakin puuttuvien testioraakkelien laatimisen vaikeus. Toisena puutteena ohjelmalle mainitaan palveluiden toimintalogiikasta johtuvat riippuvuudet: esimerkiksi poistopyynnön lähetys ilman, että poistettavaa kohdetta on perustettu aiemmin rajapinnan kautta. Näitä ei ainakaan toistaiseksi voida mallintaa mukaan testeihin.

Tsai et al. (2002) esittivät www-sovelluspalvelujen testaamiseen Coyote-mallia, jossa palveluympäristön testiskeenariot ja testitapaukset kuvataan XML-kielisellä esitystavalla. Kuvauksen perusteella testiohjelma luo varsinaiset testit erilliselle suoritusohjelmalle, joka suorittaa varsinaiset palvelupyynnöt ja tarkastaa saadut vastaukset. Mallin tarkoituksena on erottaa testaajien työhön kuuluva osuus varsinaisesta tietoliikenneosuudesta, jolloin he voivat keskittyä palvelun toiminnallisuuksien testaamiseen.

Offutt ja Xu (2004) esittävät puolestaan tapaa viestitasolla tapahtuvaan mutaatiotestaukseen. Tämä tarkoittaa viestien muokkaamista pitkälti raja-arvoanalyysiin perustuvin tekniikoin, jolloin viestin sisältöä muokataan testitapauksissa WSDL:n kuvauksen mukaisesti tietotyyppi ja mahdolliset lisärajoitteet huomioiden. Samalla periaatteella voidaan testeihin ajaa mukaan SQL-injektiohyökkäyksiä lisäämällä sopiviin sanoman kohtiin tietokannassa suoritettavia lauseita. Kolmantena mahdollisuutena menetelmässä on muodostaa testitapauksia tietosisällön rajoitteita ja riippuvuuksia varten monistamalla ja poistamalla XML-sanoman tietueita. Näin voidaan testata esimerkiksi palvelun sisältämiä duplikaattitarkastuksia tai toimintalogiikkaa toiminnallisesti virheellisen tietosisällön kera, johon WSDL-kuvaus ei kuitenkaan pysty ottamaan kantaa.

Canfora ja Di Penta (2006) nostavat palveluiden monitoroinnin testauksen apuvälineeksi. Monitorointi sisältää sekä virheiden että palvelun laadullisten tietojen, kuten vasteaikojen, kirjaamiseen liittyvien toimintojen lisäksi myös sanomaliikenteen tallentamismahdollisuuden. Tätä tietoa voidaan käyttää sekä tallennus-toistomenetelmään perustuvassa regressiotestauksessa että palveluiden jäljittelyssä testiympäristöissä. Tallennus-toistomenetelmällä tekijät tarkoittavat aiemmin tapahtuneiden palvelukutsujen toistamista palvelun uuteen versioon, jolloin uuden palvelun vastauksia voidaan verrata monitorointijärjestelmän tallentamiin vastauksiin ja ei-toiminnallisiin tietoihin. Automatisoidun toistojärjestelmän avulla testauksen laajuutta voidaan kasvattaa ilman, että palvelun loppukäyttäjien tarvitsee osallistua testaamiseen. Myös Bruno et al. (2005) ehdottavat aiempien testitapausten käyttöä palvelun uuden version testaamiseen, ja jopa testitapausten käyttämistä palvelun tarjoajan ja käyttäjän välisen sopimuksen osana, jotta molemmilla on tieto päivityksen yhteydessä suoritettavista testitapauksista.

Mei ja Zhang (2005) esittivät oman testikehyksensä, joka yhdistää oikeista kutsuista tallennetun testidatan sekä rajapintakuvauksen perusteella luotavan

aineiston käytön testaukseen. Heidän mallinsa tallentaa testiaineistoa välityspalvelinmallilla, jossa palvelukutsut ohjataan tietoja tallentavan identtisen palvelurajapinnan kautta oikealle palvelulle. Lisäksi kehys tukee laadukkaan testidatan valintaa siten, että testejä suoritaan mutaatiolla käsiteltyihin rajapintoihin. Testijoukosta pyritään valitsemaan varsinaisiin testeihin ne aineistot, jotka parhaiten havaitsevat mutaatiovirheitä, jolloin testitapausten määrää voidaan pienentää.

Palveluiden jäljittelyllä Canfora ja Di Penta (2006) tarkoittavat sitä, että palvelukutsuja testataan oikean rajapinnan sijaan tynkään, joka palauttaa aiemmin tallennettuja vastaussanomia. Tällä menetelmällä on mahdollista testata kutsuvaa järjestelmää siinä tilanteessa, jolloin palvelusta ei ole käytettävissä erillistä testiversiota.

Palveluita jäljittelevien rajapintojen automaattiseen luomiseen Bertolino et al. (2008) ovat laatineet Java-pohjaisen PUPPET-työkalun, joka on osa PLASTIC-työkalu-kokoelmaa[13]. Tämä työkalu keskittyy huomioimaan palvelutason simulointiin liittyviä tynkiä, joskin myös vastausten sisältöä voidaan sen avulla ohjata.

[13] http://plastic.isti.cnr.it/wiki/tools

5 Kohdeympäristön kuvaus

Tässä luvussa esitellään työn soveltamisen kohteena oleva asiakasympäristö ja sen yleisarkkitehtuuri. Luvussa kuvataan myös lyhyesti lähtötilanteessa käytössä olevat työkalut ja testausmenetelmät.

5.1.1 Toimintaympäristö

Merimieseläkekassa (MEK) on merenkulkijoiden lakisääteisestä työeläketurvasta huolehtiva työeläkelaitos. MEK perustettiin toimeenpanemaan vuonna 1956 voimaan astunutta merimieseläkelakia, joka oli Suomen ensimmäinen yksityisen sektorin työeläkelaki. Eläkkeiden ja avustusten maksamista varten MEK kerää vakuutetuilta työntekijöiltä sekä työnantajilta sosiaali- ja terveysministeriön vuosittain vahvistamiin maksuprosentteihin perustuvaa vakuutusmaksua. Lisäksi MEK saa eläkemenoihin valtionosuuksia, tuottoja omasta sijoitustoiminnastaan sekä mahdollisia hyvityksiä eläkeyhtiöiden välisestä vastuunjaosta. Vuonna 2008 eläkkeensaajia oli noin 8600, vakuutettuja merenkulkijoita 9200 ja työnantajina vajaat 50 varustamoa. Omaa henkilöstöä MEKillä oli 25. (MEK 2009; MEK 2010)

Suomen työeläketurva on lakisääteinen ja pakollinen. Työeläkevakuutuksia hoidetaan työeläkelaitoksissa, joita on sekä yksityisiä että julkisia. Eläketurvakeskus (ETK) on työeläkejärjestelmän yhteisiä asioita hoitava lakisääteinen yhteistyöelin, joka tuottaa myös alan yhteisiä palveluja. Yhteisten tietojärjestelmien ylläpito on keskitetty pitkälti Arek Oy:öön (Arek), jonka osakkaina on ETK:n lisäksi työeläkeyhtiöitä sekä Suomen valtio. Arek ylläpitää mm. yhteisiä eläkkeen hakemus-, päätös- ja laskentajärjestelmiä sekä keskitettyä ansaintarekisteriä, jonne talletetaan eläkkeiden perustana olevat ansaintatiedot. (ETK 2010a; ETK 2010b; Arek 2010).

Eläkeasioiden tulevaisuudesta ja muutostarpeista keskustellaan parhaillaan sekä työmarkkinoilla että poliittisesti. Työeläkeala on voimakkaasti laeilla säänneltyä, mikä tekee alasta alttiin keskustelluille muutoksille. MEK on pieni eläkelaitos, mutta sen toiminta on yhtä tarkkaan säänneltyä kuin suurempienkin toimijoiden. MEK on riippuvainen Arekin tarjoamista keskitetyistä tietojärjestelmistä, jonne käytännössä kaikki MEKin vakuutettujen ansio- ja eläketiedot siirretään. Tämä lisää MEKin riskiä järjestelmämuutoksissa, koska käytännössä isoilla toimijoilla on suurempi painoarvo päätöksissä. Arekin järjestelmien lisäksi MEK joutuu toimittamaan tietoja mm. veroviranomaisille ja pankeille. MEK onkin aloittanut vuonna 2007 järjestelmiensä uudistamisen, jonka keskeisiä tavoitteita on vaihtaa keskuskonepohjainen arkkitehtuuri joustavampaan.

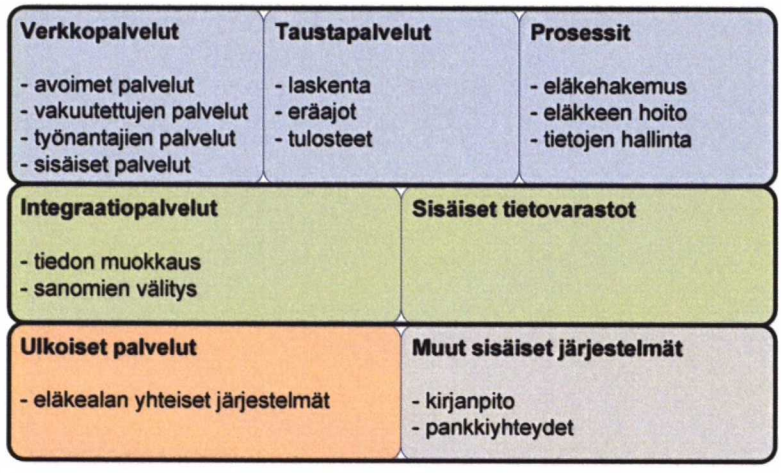
Uudistusprojektia toteuttaa Accenturen johtama projektitiimi yhdessä Avanaden kanssa. Accenture vastaa pääsääntöisesti projektin johdosta, määrittelyistä ja testauksesta Avanaden toimiessa toteuttajana ja kehitysympäristön ylläpitäjänä. Projektiryhmän koko verrattuna MEKin omaan henkilöstöön on suuri; parhaimmillaan projektissa on työskennellyt yhtä monta henkilöä kuin mitä MEKillä on omaa henkilökuntaa.

Merimieseläkekassan nykyiset keskuskonejärjestelmät on toimittanut Tieto Oyj. Tieto toimii myös MEKin palvelin- ja verkkoympäristön toimittajana.

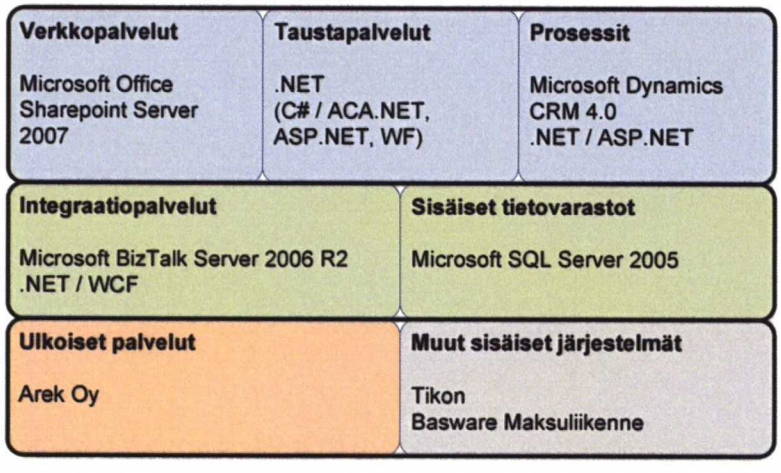
5.2 Järjestelmien yleiskuvaus ja arkkitehtuuri

Kuvassa 5.1 esitetään yksinkertaistettu kuvaus Merimieseläkekassan (MEK) uusittavista järjestelmistä. Kuvassa 5.2 on kuvattu uusittavien järjestelmien pohjaksi valitut sovellukset, tekniikat ja palveluntarjoajat. Uusia järjestelmiä toteutetaan Microsoft-ympäristöön, ja toteutuksessa hyödynnetään valmiita sovelluksia sekä joissain paikoissa kaupallisia ohjelmakirjastoja, joten ratkaisussa on riippuvuuksia ulkoisten palveluiden lisäksi valmisohjelmistokomponentteihin.

Kokonaisjärjestelmä jakautuu käsitteellisesti kolmeen erilliseen kerrokseen. Päällimmäisenä kerroksena ovat toiminnalliset kokonaisuudet, joihin kuuluvat verkkopalvelut, taustapalvelut ja sovellukset, joilla MEK hoitaa prosessejaan. Näiden alla ovat integraatio- ja tietokantapalvelut, joita liiketoimintasovellukset käyttävät. Integraatiopalveluiden kautta liiketoimintasovelluksilla on myös pääsy ulkoisiin palveluihin sekä muihin sisäisiin sovelluksiin.



Kuva 5.1: Merimieseläkekassan uusittavien järjestelmien yleiskuvaus.



Kuva 5.2: MEKin uusittavissa järjestelmissä käytetyt sovellukset ja tekniikat

5.2.1 Taustapalvelut

Taustapalveluihin kuuluvat räätälöidyt sovellukset ja palvelut, jotka suorittavat eräajoja tai yhteisesti tarvittavia toimintoja, kuten eläkkeen laskentaa tai tulosteiden muodostamista. Näitä kutsutaan sekä verkkopalveluista että asiakas-

palveluprosesseista. Taustapalveluja rakennetaan useilla .NET-tekniikoilla (ASP.NET, WCF, WF, ACA.NET). Osa palveluista on ollut tuotantokäytössä vuodesta 2008 alkaen. Järjestelmät vaativat kuitenkin säännöllistä ylläpitoa, koska esimerkiksi eläkeotteen tietosisältöä pyritään parantamaan vuosittain sekä asiakaspalautteen että uuden käytettävissä olevan tiedon perusteella.

5.2.2 Verkkopalvelut

Uusitut verkkopalvelut koostuvat neljästä eri kokonaisuudesta: kaikille avoimet verkkopalvelut, vakuutetuille ja työnantajille kohdistetut extranet-palvelut sekä sisäiset palvelut. Avoimissa verkkopalveluissa on yleistä tietoa Merimieseläkekassan toiminnasta sekä esimerkiksi lista vapaista vuokrattavista asunnoista ja liiketiloista. Tunnistautumista vaativia kohdistettuja palveluita on kahdelle eri ryhmälle: vakuutetuille sekä työnantajille. Vakuutettujen palveluissa voi käydä tutkimassa rekistereissä olevia henkilötietojaan ja sekä merimieseläkelain alaista työsuhdehistoriaansa, tehdä eläkelaskelmia sekä hakea viimeisimmän työeläkeotteensa tai todistuksen maksetusta eläkkeestä. Työnantajapalveluiden kautta työnantajavarustamot ilmoittavat maksamansa palkat ja eläkemaksut MEKille, joka puolestaan välittää tiedot työeläkealan yhteisiin tietojärjestelmiin ja viranomaisille. Sisäiset palvelut on kohdistettu Merimieseläkekassan henkilökunnan sekä hallinnon käyttöön. Palvelu perustuu työntekijäkohtaisiin rooleihin, joiden perusteella työntekijä saa nähtävilleen työnkuvaansa eniten koskettavat työkalut ja dokumentit. Sisäinen palvelu toimii myös henkilökunnan tiedonvälityskanavana. Verkkopalvelujen viimeinen suuri kokonaisuus uusittiin helmikuussa 2010 ja ne ovat nyt ylläpitovaiheessa. Palvelut on toteutettu käyttäen Office Sharepoint Server 2007 (MOSS) alustaa, jonka päälle on tehty tarvittavat räätälöinnit .NET-tekniikoilla.

5.2.3 Prosessit

Asiakaspalveluprosessien hoitamisen tarvitsemat sovellukset toteutetaan asiakkuudenhallintaohjelmiston (Microsoft Dynamics CRM 4.0) päälle räätälöidyillä ratkaisuilla. Tärkeimmät prosessit toteutettavassa järjestelmässä ovat eläkehakemuksen käsittely sekä eläkkeen hoito, jonne hyväksytysti käsitellyn eläkehakemuksen tuloksena syntyy uusi maksettava eläke. Hoitojärjestelmään pitää käyttöönoton yhteydessä yhdistää aiemmin hyväksytyt eläkkeet. CRM-järjestelmässä ylläpidetään myös vakuutettujen, eläkeläisten sekä sidosryhmien yhteystietoja toiminnan kannalta tarpeellisella tarkkuudella. Toteutuksessa hyödynnetään CRM-ohjelmiston tarjoamaa käyttöliittymää, tietomallia ja sovelluslogiikkaa mahdollisimman paljon. Osa toiminnoista pitää toteuttaa räätälöidyillä laajennuksilla, jotka upotetaan CRM:n sisään omina sivuinaan tai IFRAME-kehyksiä käyttäen. Räätälöinneissä pyritään noudattamaan kerros-arkkitehtuuria kuvassa 5.3 esitetyllä tavalla. Lomakkeiden tietoja validoidaan myös JavaScriptiä käyttäen, mikä on CRM:n sisäinen toimintamalli. Hakemus- ja hoitojärjestelmän uusiminen on käynnissä parhaillaan, ja niiden käyttöönotto on aikataulutettu vuosin 2010 ja 2011 vaihteeseen.

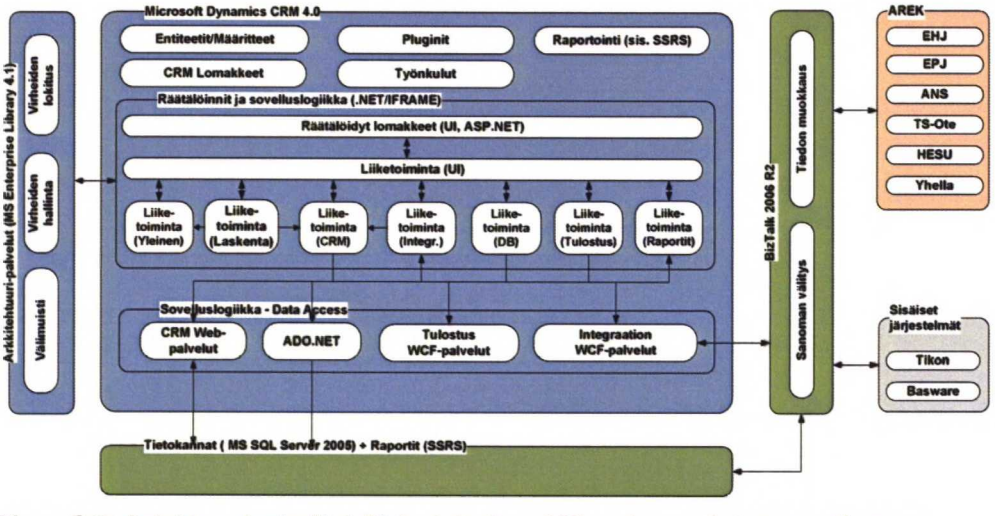
5.2.4 Integraatiopalvelut, tietokannat ja ulkoiset liittymät

Integraatiopalvelut välittävät palvelupyyntöjä muiden sisäisten ja ulkoisten järjestelmien välillä huolehtien myös tarvittavista sanomamuunnoksista. Ulkoiset palvelut ovat keskittyneet Arek Oy:n palveluihin. Arekin palvelukokonaisuuksia ovat mm. eläkepäätös- (EPJ), eläkehakemus- (EHJ), ansainta- (ANS), työsuhdeote- (TS-Ote), henkilötieto- (HESU) ja yhteinen eläkkeenlaskenta-

järjestelmä (Yhella). Arekin liittymät ovat pääosin www-sovelluspalveluita (SOAP) sekä FTP-siirtojen kautta XML-muotoisilla aineistoilla tapahtuvia eräajoja. Kaikkia palveluita ei kuitenkaan tarjota vielä näiden rajapintojen kautta, joten siirtymävaiheessa MEK joutuu toteuttamaan liittymiä myös IBM:n MQSeries sanomajonojen kautta peräkkäistiedostomuotoisena. Kaiken kaikkiaan erilaisia sanomia tulee olemaan käyttöönottoon mennessä noin 30 kpl, mutta yhdellä sanomalla voi olla useita erilaisia käyttötapoja ja toiminnallisuuksia. Esimerkiksi tietojen mitätöinti ja uusien tietojen rekisteröinti voi tapahtua saman sanoman avulla. Osa liittymistä on erittäin monimutkaisia. Esimerkiksi peräkkäistiedostomuotoisen yhteisen eläkkeen laskentapalvelun vastaussanoman BizTalk-skeeman koko on noin 6 megatavua sisältäen yli 6500 tietueen kuvauksen. Palvelujen testaaminen voi vaatia myös usean sanoman siirtoa, koska palvelut toteuttavat kokonaisprosessia – esimerkiksi eläkehakemuksen mitätöinti vaatii ensin hakemuksen luomisen Arekin järjestelmään. Sisäisten järjestelmien kanssa kommunikointi tapahtuu tiedostopohjaisesti sekä XML- että peräkkäis-tiedostomuodossa.

Sisäisten tietovarastojen, lähinnä SQL Server -tietokantojen, käyttö voi tapahtua suoraan sovelluksesta tai integraatiopalveluiden kautta. Tärkeimmät erilliset sisäiset sovellukset ovat kirjanpito ja pankkiyhteydet. Integraatiopalveluita tarjotaan BizTalk Server 2006 R2:n sekä räätälöityjen, lähinnä .NET/WCF-teknikkaa käyttävien www-sovelluspalveluiden kautta.

Merimieseläkekassan ympäristöä ei voi pitää sinänsä palvelusuuntautuneena arkkitehtuurina (SOA), vaikka liiketoiminta- ja integraatiokerroksessa onkin paljon www-sovelluspalveluihin perustuvia rajapintoja. Tämä johtuu siitä, ettei ympäristöä kehitetä tietoisesti ja kokonaisvaltaisesti SOA-mallin mukaan. MEKin organisaation ja toiminnan pienuudesta johtuen tämä olisi aivan liian raskas toimintatapa. Kehitystyössä yritetään kuitenkin tunnistaa yleiskäyttöisiä komponentteja tai prosesseja, joista voisi olla hyötyä muillekin komponenteille nyt tai tulevaisuudessa. Tällaiset palvelut pyritään avaamaan palvelurajapinnan kautta muidenkin käyttöön, mutta palveluiden tunnistamista ja suunnittelua ei tehdä liiketoimintalähtöisesti, kuten "aito" palvelusuuntautuneisuus edellyttäisi. Myöskään palveluhakemistoja ei käytetä. Järjestelmän testaamiseen liittyviin periaatteisiin palvelusuuntautunut ajatusmalli kuitenkin sopii.



Kuva 5.3: Asiakkuudenhallintajärjestelmän päälle rakennettujen sovellusten yleisarkkitehtuuri (kuva laadittu yhteistyönä MEK-projektiryhmän kanssa)

5.3 Muutostarpeita aiheuttavia tekijöitä

MEKin järjestelmäympäristöön eniten vaikuttava tekijä on Arekin järjestelmien kehitysrytmi. Palveluista julkaistaan uusia versioita useita kertoja vuodessa. Tämä aiheuttaa jatkuvaa ylläpitotarvetta MEKin järjestelmille niin tuotannon kuin kehityksenkin osalta. Jatkossa MEK osallistuu laajemmin Arekin hyväksymistestausprosessiin, minkä vuoksi liittymiä tulee voida testata myös suoraan ilman integraatiokerroksen käyttöä. Liittymiä käyttävät MEKin sovellukset tulisi regressiotestata aina Arekin päivitysten yhteydessä. Arekin liittymämuutoksista johtuva regressiotestaus on arvioitu suurimmaksi ylläpitotyötä aiheuttavaksi tekijäksi.

Hakemus- ja hoitojärjestelmää kehitetään iteratiivisesti, minkä vuoksi kehitystyön aikana on varauduttava muutoksiin ja uudelleentestaukseen. Osa hakemusprosessin toiminnallisuuksista on tietoisesti jätetty toteutettavaksi vasta kokonaisjärjestelmän käyttöönoton jälkeen, mutta iteratiivisuuden vuoksi tästä ei odoteta suuria toiminnallisia muutoksia. Ylläpitovaiheessa tullaan muuttamaan kaikille prosesseille yhteisten henkilö- ja yritystietojen hallintaa perinteisen asiakkuudenhallinnan näkökulmasta. Tällä voi olla vaikutusta myös hakemusjärjestelmään siltä osin kuin ne käyttävät henkilöiden perustietoja.

Osa ensimmäisten vaiheiden järjestelmistä piti rakentaa käyttämään keskuskoneen tarjoamia rajapintoja. Näiden liittymien kohdalla keskuskoneen nykyiset toiminnot pitää korvata omilla toteutuksilla tai Arekin palveluita yhdistämällä syksyn 2010 aikana. Liittymärajapinnat halutaan kuitenkin pitää ennallaan, jottei itse sovelluksia tarvitse muuttaa. Tässä yhteydessä on oleellista varmistaa, etteivät muutokset riko mitään. Käytännössä liittymien tulee toimia muuttumattomina, joten niiden toiminnan varmistaminen voidaan tehdä vertailemalla vanhoja ja uusia sanomia.

Luvussa 5.1.1 mainitulla tavalla eläkelakien muutokset voivat aiheuttaa muutostarvetta. Lakimuutokset ovat kuitenkin harvinaisempia, ja MEK osallistuu niiden valmistelutyöhön, joten näiden muutosvaikutukset tiedetään hyvissä ajoin. Lakimuutokset voivat aiheuttaa järjestelmän liiketoimintalogiikkaan suuria toiminnallisia muutoksia, joten niiden yhteydessä myös useat testitapaukset vanhenevat.

5.4 Testauksen ja kehitysympäristön nykytila

Tässä luvussa käsitellään testauksen ja kehitysympäristön nykytilaa mukaan lukien kehitys- ja testaustyökalujen käyttö. Luvun laatimisessa on käytetty hyväksi ei-julkisia projekti- (MSKPS 2010) ja testaussuunnitelmia (MSKTS 2010) sekä projektin henkilöstön kommentteja.

5.4.1 Kehitys- ja testausympäristö sekä työkalut

Vaatimusten, testitapausten, vikojen (defektien) sekä lopputuotteiden katselmointien tiedot talletetaan selainliittymän kautta toimiville Rational-sovelluksille. Vaatimukset talletetaan RequisitePro-järjestelmään. Hoitojärjestelmän testitapaukset talletetaan Rational Quality Manager -järjestelmään (RQM). Hakemusjärjestelmän testitapaukset ovat erillään dokumentinhallintajärjestelmässä, koska RQM ei ollut käytettävissä hakemusjärjestelmän toteutuksen alussa. Defektit ja lopputuotteet katselmointikommentteineen kirjataan ClearQuest-järjestelmään.

Ohjelmistokehitys tapahtuu asiakkaan vuokraamille palvelimille asennetuissa virtuaalikoneympäristöissä. Kehitysympäristö sisältää myös järjestelmätestin tarvitsemat palvelimet, minkä lisäksi käytössä ovat erilliset hyväksymis- ja tuotantotestiympäristöt, joissa myös asiakas osallistuu testaamiseen. Hyväksymistestissä testataan uusia järjestelmiä kehitysvaiheen aikana. Tuotantotestissä suoritetaan tuotantokäytössä olevien ohjelmistojen päivitysversioiden testausta. Sen kautta tehdään myös uusien sovellusten testaus ennen niiden tuotantoonsiirtoa. Tällä pyritään varmistamaan, ettei käyttöönotto aiheuta ongelmia jo tuotannossa oleville järjestelmille.

Kehitystyökaluina ovat Visual Studion versiot 2008 ja 2005, jota käytetään lähinnä vain BizTalk-kehitykseen. Versionhallinta toimii Microsoftin Team Foundation Serverillä (TFS), mutta muita TFS:n toimintoja ohjelmiston elinkaaren hallintaan ei käytetä aktiivisesti. Linkitystä TFS:n ja Rational-järjestelmien tietoihin ei ole voitu rakentaa, joten vaatimusten, testitapauksien ja defektien tietojen linkitys ohjelmakoodiin ja sen muutoksiin katkeaa. Kaksinkertaista kirjaamista molempiin järjestelmiin ei tehdä.

Hakemus- ja hoitojärjestelmän julkaisua varten on erillinen käännöspalvelin. Uusien versioiden julkaisut tehdään toistaiseksi manuaalisesti, mutta käännösten ja yksikkötestien automatisointi tällä palvelimella on jo aloitettu kehitystiimin toimesta. Yksikkötestauksessa käytetään Visual Studion sisältämiä testityökaluja. Liittymätestauksessa ajurina ja tynkänä käytetään myös soapUI:ta sekä itse laadittuja WCF-testausohjelmia. soapUI:n käyttö on kuitenkin rajoittunutta, koska se ei tue WCF-rajapintoja.

5.4.2 Testaustavat

Projektissa yksikkötestaus on kehittäjien vastuulla. Kehitystapa ei ole testauslähtöinen, joten testin toteutus ja suoritus tapahtuu yleensä komponentin toteutuksen jälkeen. Testauksen laajuutta ei ole määritelty sopimuksissa tai testaussuunnitelmassa tarkoilla mittareilla. Yksikkötestauksessa ei pyritä aina komponentin täydelliseen eristämiseen, koska järjestelmä toimii valmisohjelmiston päällä, vaan testi alustaa ensin CRM-järjestelmän tietokantaan tarvitsemansa pohjatiedot. Yksikkötestauksessa hyödynnetään myös aiemmin testattuja komponentteja tynkien ja ajureiden sijaan Myersin kuvaamalla inkrementaalisella mallilla (ks. luku 2.4.3), koska osa tyngistä olisi liian monimutkaisia ja siten suuritöisiä ja virheherkkiä. Tämän lisäksi varsinaista integraatiotestausta ei tehdä suunnitelmallisesti, vaan tarpeen mukaan virhetilanteiden selvityksen tueksi, mikäli kokonaisuuden toiminnassa havaitaan ongelmia.

Järjestelmätestaus tehdään suunnitelmallisesti iteraatioittain. Erillinen testitiimi laatii testitapaukset pääasiassa käyttötapausten avulla ja päivittää niitä sitä mukaa, kun muutoksia syntyy. Testit suunnitellaan toiminnallisin menetelmin, lähinnä raja-arvoanalyysillä sekä virhekohtia testaajan toimesta valistuneesti arvaamalla (katso luku 2.3.1). Käyttötapauksissa on linkitys järjestelmän vaatimuksiin, joten testitapaukset saadaan linkitettyä näihin molempiin. Jokaiselle järjestelmän osalle pyritään suorittamaan vähintään kaksi testikierrosta. Suunniteltujen testitapausten lisäksi testaajat suorittavat kokeellista testausta. Järjestelmätestauksen aloituskriteerinä on, että testitapaukset ja -aineistot on katselmoitu ja hyväksytty, ja että testattava sovellus on asennettu testiympäristöön ja yksikkötestit on suoritettu onnistuneesti. Asiakas suorittaa hyväksymistestausta iteraatioittain ja sekä koko järjestelmän testauksen erikseen vaiheen päättyessä.

Iteratiivisesta kehitysmallista sekä Arekin rajapintamuutoksista johtuen kehityksen aikana tarvitaan regressiotestausta. Regressiotestaukseksi lasketaan myös uuden kehitysversion savutestaus, joka suoritetaan järjestelmätestitapauksien avulla. Testaajat valitsevat tarvittavat testitapaukset regressiotesteihin oman osaamisensa ja kehittäjiltä saamiensa tietojen perusteella.

Liittymärajapintojen testaus suunnitellaan erikseen. Testaus suoritetaan integraatiotestauksen luonteisesti suorittamalla kutsuja liittymäkohtaisten testiohjelmien avulla ensin alempiin sovelluskerroksiin kunnes lopulta koko ketju testataan järjestelmätestitapauksella käyttöliittymän kautta. Tällä varmistetaan, että järjestelmä osaa myös näyttää vastaussanomasta tarvittavat tiedot käyttöliittymässä oikein. Ulkoisten liittymien osalta tarvittavat testiaineistot järjestetään MEKin avustuksella Arekin ympäristöihin, joita on useita. Testaajat ylläpitävät listaa eri ympäristöjen testihenkilöistä. Arekin palveluiden käyttö voi aiheuttaa testihenkilöiden tietojen muutoksia, joten tietosisällölle herkkien testitapausten tarvitsemia testihenkilöitä ei saa käyttää muutoksia aiheuttavissa testeissä.

Verkkopalvelukomponenttien testaaminen tapahtuu lähinnä järjestelmätestauksen tasolla. Jos komponenttiin liittyy erillisiä luokkakirjastoja, näiden yksikkötestaus on mahdollista, mutta MOSS-alustaan liittyvien toimintojen testaaminen irrallaan on toistaiseksi kehitystyökalujen puolelta huonosti tuettu toimenpide.

5.4.3 Katselmoinnit

Kaikki lopputuotteet määrittelydokumenteista testitapauksiin ja ohjelmakoodiin asti pyritään katselmoimaan vertaisarvioinneilla, mutta aikataulusyistä tämä jää joskus myöhäisempään vaiheeseen kuin mikä olisi virheiden havaitsemisen kannalta hyödyllistä. Tämä on yhtenevää Harjumaa et al. (2006) havaintojen kanssa (katso luku 2.2.2).

Kehitystiimi on ottanut työn aikana koodin läpikäynnit ohjelmaan. Näiden tarkoituksena on laadunvarmistuksen lisäksi levittää tietämystä ja yhtenäistää ohjelmointikäytäntöjä kehittäjien kesken. Projektissa on järjestetty myös sisäisiä auditointeja, joilla pyritään varmistamaan suunnitelmien laatu ja toteutuskelpoisuus.

5.4.4 Mittarit

Defekteistä kirjataan päivämäärän ja ongelmakuvausten lisäksi ylös virheen kriittisyys neliportaisella asteikolla. Testitapauksen numero kirjataan RQM-järjestelmään, mutta vanhoista defekteistä tietoa ei välttämättä ole saatavilla. Vian korjaamisen yhteydessä järjestelmään syötetään kuvaus sen syystä ja tehdystä korjauksesta, mutta selkeää luokittelua tai linkitystä häiriön aiheuttaneeseen komponenttiin ei tallenneta. Kehittäjiä on ohjeistettu syöttämään defektin tunnus versionhallinnan vapaamuotoiseen kommenttiin vian korjausta tallennettaessa, mutta tätä ei voida nykyisellään pakottaa, eikä vapaamuotoista tietoa ole helppo käyttää analysointiin.

Testien kattavuutta ei mitata tai seurata säännöllisesti järjestelmätestien vaatimusten kattavuutta lukuun ottamatta. Testitapauksen suorittamiseen kuluvaa aikaa ei myöskään eritellä, vaan testaajat kirjaavat työaikansa projektisuunnitelmasta löytyville ylemmän tason tehtäville.

Testauksen tilanteesta ajetaan viikoittain raporteille testien läpimenoprosentit niin testikierroksen kuin kokonaisuuden osalta. Viikkoraportilla on myös avoimien defektien määrä luokiteltuna kriittisyyden sekä osajärjestelmän suhteen.

Avoimeksi defektiksi lasketaan myös testaamaton, mutta korjatuksi ilmoitettu defekti. Defektien määrää seurataan myös pidemmällä aikavälillä niiden tilan mukaan. Testauksen viikkoraportoinnin yhteydessä suoritetaan myös riskienhallintaa, jonka yhteydessä käydään läpi testauksen perusteella havaittuja asioita ja mietitään mahdollisuuksia riskien pienentämiseksi. Versionhallinnassa olevasta koodista ei ajeta säännöllisiä raportteja esimerkiksi rivimäärän tai muutosten määrän kehittymisen suhteen.

5.4.5 Testauksen ongelmakohtia

Hakemusjärjestelmän regressiotestaus on osoittautunut erittäin suuritöiseksi, joka vaati aiemmilla testikierroksilla yhden testaajan työpanoksen lähes koko iteraation ajalta. Hakemusprosessin kompleksisuudesta johtuen jopa savutestiksi suunniteltu hakemuksen peruskäsittely parilla variaatiolla on laaja testi. Hakemusprosessi sisältää yhden kriittisen toiminnon, eläkeiän laskennan, jota ilman mitään testitapausta ei voida suorittaa loppuun asti.

Resurssipulan vuoksi suunnitellun laajuisesta regressiotestauksesta on jouduttu toistaiseksi luopumaan. Testaus keskittyy nyt korjattujen vikojen uudelleentestaukseen. Tässä yhteydessä suoritetaan joitain testejä myös regressiotestijoukosta. Savutestien automatisointia tallennus-toistotyökaluilla on mietitty Visual Studio Test Editionin perustesteillä, mutta kehitysversion epävakaudesta johtuen automatisointia ei ole vielä tehty. Hakemusjärjestelmän kehitysversioissa on esiintynyt jonkin verran regressiovirheitä.

Arekin rajapintamuutoksista aiheutuva muutosten analysointi ja testaus on haasteellista muutamien liittymien monimutkaisuudesta johtuen. Tämä aiheuttaa ongelmia sekä integraatiopalveluissa että sovelluksen liiketoimintalogiikassa. Osa monimutkaisista ulkoisista liittymistä on melko tiukasti sidoksissa toteutukseen ja niiden dokumentaatiovirheet aiheuttavat usein vasta testeissä havaittavia häiriöitä. Arekin testiympäristöissä on myös paljon katkoksia, joista vain osa on suunnitelmallisia ja ennakkoon tiedossa. Testaajien kannalta virheilmoitukset eivät kerro aina riittävän selkeästi, onko kyse ulkoisen palvelun katkoksesta vai testattavan sovelluksen tai integraatiopalveluiden aiheuttamasta ongelmasta. Pelkästään ulkoisten palveluiden katkoksiin kuluu testaajilta turhaa työaikaa useita tunteja viikossa.

Testaajat ovat toivoneet tarkempia tietoja siitä, mitä muutoksia uuteen testiversioon on tullut edelliseen verrattuna. Tiedotus tapahtuu nyt täysin manuaalisesti lyhyiden julkaisukommenttien kautta. TFS:n työlistoja ei käytetä. Näin osa muutoksista jää usein raportoimatta, eikä testaajilla ole mahdollisuutta päästä käsiksi muutostietoihin itse.

Hoitojärjestelmän osalta käyttöliittymätestaus on yksinkertaisempaa, koska lomakkeet sisältävät vähemmän älykkyyttä ja prosessit ovat suoraviivaisempia kuin hakemusjärjestelmässä. Hoitopuolella kriittisimmät testauskohteet liittyvät tietokannan sekä liittymäaineistojen, esimerkiksi maksuerien, oikeellisuuden tarkastuksiin. Näiden silmämääräinen vertailu molemmista paikoista on vaikeaa, eikä testejä ole vielä automatisoitu.

6 Ehdotetut muutokset

Tässä luvussa esitellään ehdotetut muutokset testaus- ja kehitysprosessiin perusteluineen. Lisäksi esitellään tarkemmat suunnitelmat toteutettavien apuohjelmien toiminnallisuudesta ja rakenteesta.

6.1 Yleistä

Työn tavoitteena oli suunnitella ratkaisumalleja Merimieseläkekassan uudistettavien järjestelmien ylläpitovaiheen tehostamiseksi. Testaus on projektin kehitysvaiheessa kuormittava työvaihe, johon on myös varattu resursseja (ks. luku 5.4.2 ja 5.4.5). Ylläpitovaiheessa yhtä suurta testitiimiä ei ole käytettävissä, minkä vuoksi testausta tulee tehostaa, jottei sen kattavuudesta tarvitse tinkiä. Testauksen tehostaminen jo kehitysvaiheessa vapauttaisi resursseja rutiinityöstä laadukkaampaan testaamiseen ennen hakemus- ja hoitojärjestelmän käyttöönottoa. Näin tehostamistoimista voitaisiin saada kustannushyötyjä jo ennen ylläpitovaihetta.

Työssä haettiin vastauksia etenkin luvussa 1.2 esiteltyihin projektin aikana nousseisiin keskeisiin ongelmakohtiin. Palvelupyyntöjen ja niistä muodostuvien prosessien tehostamista käsitellään luvussa 6.2, ulkoisten rajapintojen katkosten vaikutusten pienentämistä luvussa 6.3, palvelurajapintamuutosten vaikutusten analysointia luvussa 6.4. Näitä ongelmia tutkiessa syntyi myös muita kehitysehdotuksia, joita on käsitelty luvussa 6.5. Saman luvun alla käsitellään myös testien määrän vähentämistä, joka oli neljäs keskeinen projektin aikana esiin noussut ongelmakohta. Osa ehdotuksista on tarkoitettu hyödynnettäväksi vasta mahdollisissa jatkoprojekteissa tai muissa ympäristöltään samankaltaisissa uusissa projekteissa, koska niiden käyttöönotto ei ole järkevää toteutuksen keskellä.

Työn alkuvaiheessa palvelurajapintojen testauksen mahdollisimman pitkälle viety automatisointi vaikutti keskeisimmältä ratkaisulta, mutta kirjallisuuskatsauksen aikana paljastui kuitenkin useita seikkoja, joiden parantaminen olisi järkevämpää kuin pelkän automatisoinnin toteuttaminen. Kuten testiautomaatiota käsittelevissä luvuissa 3.3 ja 3.4 kerrottiin, automatisointi kannattaa toteuttaa vähitellen ja huolehtia siitä, että muut testaukseen liittyvät asiat ovat ensin riittävän hyvässä kunnossa.

Kokonaisprosessin automatisoinnin sijaan ehdotetuissa ratkaisuissa tehostamista haetaan useiden pienempien toimenpiteiden kautta. Osa näistä käsittää myös testien suorituksen automatisointia, ja näitä muutoksia voidaan pitää pilottiprojekteina ja ensimmäisinä vaiheina testiautomaation kehittämisessä. Luvussa 3.4 mainituista syistä rauhallinen etenemistapa on edellytys testiautomaation onnistumiselle, joten se on perusteltu lähtökohta tässäkin projektissa.

6.2 Palvelupyyntöjen testaamisen tehostaminen

6.2.1 Www-sovelluspalveluiden testaustyökalu – BizUnit

Hakemus- ja hoitojärjestelmän liiketoimintakerros sisältää WCF-palvelurajapintoja, joihin voidaan periaatteessa julkaista mikä tahansa liiketoimintakerroksen sisältämä toiminnallisuus. Sekä WCF-rajapintoja että perinteisiä SOAP-kutsuja tukeva testiohjelma mahdollistaisi järjestelmän testaamisen monipuolisesti käyttöliittymäkerroksen alla, mikäli sillä voisi lähettää vapaamuotoisia XML-aineistoja. Lisäksi sillä voitaisiin testata sekä integraatiopalveluiden että Arekin

liittymiä. Nyt testausohjelmia ja testien vaatimia ajureita rakennetaan liittymäkohtaisesti. Yleiskäyttöisellä testiohjelmalla tämä toistuva työ voitaisiin välttää. Lisäksi työkalu mahdollistaisi aiemmista palvelukutsuista tallennetun aineiston toistamisen regressiotestauksessa. Aineistojen tallennusta käsitellään lisää luvussa 6.3.

Edellä mainittua testausmallia ja työkalua tukevat testien automatisoinnin kannalta luvussa 3.4 mainitut suositukset testien ajamisesta ohjelmarajapintojen kautta ja tarvittavia testimetodeita lisäten (Kaner et al. 2002; Martin 2005). Berner et al. (2005) kommentoi lisäksi, että hyvä testiautomaatio edellyttää eri testaustasojen hyödyntämistä. Heidän tutkimuksessaan erään esitellyn projektin testiautomaatio oli keskittynyt nimenomaan käyttöliittymän alapuolisten toimintojen testaamiseen. Tämä työkalu mahdollistaisi saman lähestymistavan testien automatisoinnille ja samalla integraatiotestauksen laajentamisen, mitä MEK-projektissa suoritetaan selvästi vähemmän kuin muuta testausta. Tallennettujen sanomien toistamista regressiotestauksen yhteydessä ehdottivat luvussa 4.3.1 mainitut Canfora ja Di Penta (2006), Bruno et al. (2005) sekä Mei ja Zhang (2005).

Tarvittavan testiohjelman pitäisi tukea joko käyttöliittymässä näytettävän tai erilliseen tiedostoon talletetun XML-muotoisen sanoman lähettämistä palvelurajapintaan. Myös lähetysosoite pitää pystyä konfiguroimaan vastaavalla tavalla. Kolmantena toivottavana vaatimuksena olisi mahdollisuus ketjuttaa palvelukutsuja kokonaisprosesseiksi, mikä edellyttää mahdollisuutta suorittaa aineistoille tarkastuksia ja muokata lähetettäviä kutsuja edellisten vastausten sisällön perusteella. Neljäs tarvittava ominaisuus on mahdollisuus testien automaattiseen suorittamiseen. Mahdollisuus testitapausten kuvaamiseksi avainsanapohjaisesti olisi toivottava lisäominaisuus, jotta työkalun käyttäminen olisi luvussa 3.5 mainitulla tavalla testaajille mahdollisimman luontevaa.

Valmisohjelmien evaluointi

Www-sovelluspalvelutestejä varten käytössä oleva soapUI-sovellus ei tue WCF-rajapintoja kuin basicHttpBindingin kautta. Vaikka sovellus on muuten monipuolinen ja tukee skriptien avulla hyvinkin monimutkaisia toimintoja mukaan lukien palvelurajapintojen jäljittely, ei ohjelma tämän puutteen vuoksi kelpaa MEKin kehitysympäristössä kokonaisratkaisuksi. Sovellukseen on suunnitteilla WCF-tuki, mutta sen aikataulusta ei ole minkäänlaista tietoa.

Vastaavaa WCF-rajapintoja tukevaa sovellusta ei suoraan löydy. Visual Studion mukana toimitettava WcfTestClient on tarkoitettu ainoastaan yksinkertaisten palvelukutsujen lähettämiseen, eikä se tue edes lähetettävien aineistojen lukemista tiedostoista.

Kaupallinen WcfStorm[14] tukee yksittäisten palveluiden testaamista, vastausten tarkistuksia sekä kuormitustestausta, mutta se ei tue prosessien testaamista. Suoraa tukea tieto- tai avainsanapohjaiselle testaamiselle ei ole, mutta laajennusrajapinnan kautta tämä on mahdollista toteuttaa. WcfStormin tarkastusmahdollisuudet ovat valmiinakin kohtuulliset, minkä lisäksi omien laajennusten käyttöä tuetaan. Toistaiseksi ohjelma ei tue testien automatisointia komentorivityökalun avulla, mutta tämä ominaisuus on merkitty seuraavassa versiossa julkaistavaksi.

[14] http://www.wcfstorm.com/

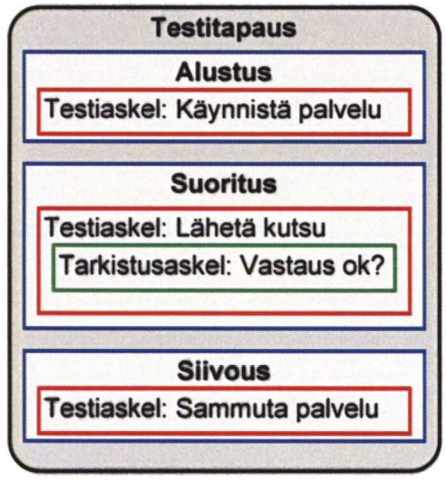
Ulkoasultaan yksinkertainen SOA Cleaner[15] mahdollistaa useiden palveluiden ketjuttamisen ja edellisten vaiheiden vastausten käyttämisen uuden pyynnön muodostamisessa. Pyyntöjä on mahdollista toteuttaa myös tietopohjaisesti tai arvoja satunnaistamalla. Monivaiheisen testin askelien väliin on mahdollista asettaa ehtoja. SOA Cleaner osaa myös käsitellä poikkeustilanteita testitapauksina. Testien automatisointi on mahdollista komentorivityökalun avulla.

Molemmat edelliset ohjelmat tallettavat testitapaukset ja syötteet XML-muotoon, joten niitä manipuloimalla monimutkaisempien testitapauksien rakentaminen olisi mahdollista, mutta vaatisi itse tehtyjä työkaluja. Tämä olisi myös virheherkkä tapa toimia. SOA Cleanerin kanssa malli toimisi paremmin, koska se tukee komentorivikäynnistystä ja tallentaa lokiin kaikki pyyntö- ja vastaussanomat kokonaisuudessaan.

BizUnitin evaluointi

Näiden graafisten työkalujen ulkopuolelta tutkittiin BizTalk-testauksessa suosittua BizUnit-työkalua[16], jota oli evaluoitu aiemminkin integraatiopalveluiden testausta suunniteltaessa. Tuolloin sen ominaisuudet havaittiin puutteellisiksi, eikä sitä otettu laajemmin käyttöön. Nimestään huolimatta sen käyttömahdollisuudet eivät rajaudu BizTalk-sovelluksen testaamiseen, sillä monet ohjelman sisältämät toiminnallisuudet ovat yleiskäyttöisiä. Näitä ovat esimerkiksi tiedoston kopioinnit ja lukemiset halutusta paikasta, SOAP- tai HTTP-kutsujen lähetys ja XML-aineiston validointi.

BizUnit toimii suoraan avainsanapohjaisesti. Siihen on toteutettu lukuisia toimintoja, ns. *testiaskelia*, joiden avulla testitapaukset kuvataan XML-tiedostoon. Testitapaus muodostuu testin alustuksesta, suorituksesta ja siivouksesta kuvassa 6.1 esitetyllä tavalla. Jokainen vaihe voi sisältää useita askeleita, joita voidaan suorittaa myös rinnakkaisesti. Testiaskelen sisään määritetään sen toimintokohtaiset parametrit. Kuvassa 6.2 on esimerkki BizUnit-testitapauksesta, joka suorittaa www-sovelluspalvelukutsun ja tarkastaa sen vastauksen ilman alustus- tai siivoustoimenpiteitä. Testitapausten XML-muoto mahdollistaa myös niiden luomisen muista työkaluista suhteellisen yksinkertaisin menetelmin.



Kuva 6.1: BizUnitin testitapauksen rakenne

[15] http://xyrow.com/
[16] http://bizunit.codeplex.com/

```
<TestCase testName="SoapRequestResponseTest">
    <TestExecution>
        <TestStep assemblyPath="" typeName="BizUnit.SOAPHTTPRequestResponseStep">
            <WebServiceWSDLURL>http://localhost/Ankkuri_BTS_WS_Korjaukset/Korjaukset.asmx?wsdl</WebServiceWSDLURL>
            <ServiceName>Korjaukset</ServiceName>
            <WebMethod>MELSUKorjaus</WebMethod>
            <InputMessageTypeName>KorjausWS_MELSU</InputMessageTypeName>
            <MessagePayload>D:\DEV_Esa\BizUnitTest\BizUnitTest\MELSU.xml</MessagePayload>
            <ValidationStep assemblyPath="" typeName="BizUnit.XmlValidationStep">
                <XPathList>
                    <XPathValidation query="/*[local-name()='Vastaus']/*[local-name()='Onnistui']">true</XPathValidation>
                </XPathList>
            </ValidationStep>
        </TestStep>
    </TestExecution>
</TestCase>
```

Kuva 6.2: Esimerkki BizUnit-testitapauksesta

Kokonaisuutena BizUnitin toimintamallissa on paljon samoja piirteitä kuin luvussa 4.3.1 mainitussa www-sovelluspalveluiden testaamiseen tarkoitetussa Coyote-testikehyksessä (Tsai et al. 2002) lukuun ottamatta testiaineistojen luomista.

BizUnit sisältää valmiina tuen SOAP-kutsujen tekoon sanoman sisällön XML-osan avulla. Tämä edellyttää kuitenkin, että palvelu luovuttaa WSDL-kuvauksen suoraan kutsuosoitteesta, minkä perusteella testiohjelma luo kutsun vaatiman ns. *proxy-luokan* ajon aikana. Arekin palvelut eivät toimi näin, vaan kuvaus toimitetaan aina erillisessä tiedostossa. SOAP-kutsujen suoritus on kuitenkin mahdollista myös suorana HTTP-pyyntönä, jolloin sanomana välitetään koko SOAP-kutsu kuorirakenteineen.

WCF-rajapintoja käyttävä testiaskel on saatavina laajennuksena[17] verkosta. Laajennus toimii eri logiikalla kuin edellinen SOAP-kutsu, koska se vaatii proxy-luokan kääntämisen valmiiksi ennen testien suorittamista. Tämä ei ole sinänsä vaikea operaatio, mutta vaikeuttaa ohjelman käyttöä perustestaukseen ja vaatii käytännössä kehittäjän toimenpiteitä. Laajennuksen toisena ongelmana voi pitää sinänsä WCF-mallin mukaista käytäntöä lukea yhteysparametrit konfiguraatiotiedostosta. Loogisempaa olisi yhdistää nämä tiedot testitapaukseen tai ainakin mahdollistaa molemmat konfigurointitavat.

BizUnit tukee vastaussanomien arvojen lukemista muistiin ja niiden käyttöä testin myöhemmissä vaiheissa esimerkiksi pyyntösanoman muokkaamiseen ns. Context-objektin kautta. Tämä mahdollistaa palveluiden ketjuttamisen. Testien suorittamiseen voidaan käyttää lähes mitä tahansa kehikkoa, joten suorituksen automatisointi onnistuu esim. Visual Studion sisältämän MsTest-komentorivityökalun tai erillisen NUnit-kehikon avulla. Graafisen käyttöliittymän puuttuessa testaajia helpottaa BizUnitille tehty laajennus testiskriptien syöttämiseksi Excel-taulukoiden kautta. XML-aineistojen muokkaamiseen on saatavilla Visual Studion lisäksi myös erillisiä työkaluja, ja niiden käyttäminen on myös testaajille tuttua.

Tarvittavat laajennukset

BizUnit laajennuksineen tarjosi nykyisellä 3.1-versiollaan hieman yllättäen BizTalk-sovelluksen yksikkötestauksen lisäksi parhaat valmiudet myös palveluiden ja palveluprosessien testaamiseen, varsinkin kun BizUnitin käyttö ei edellytä BizTalk Server -tuotteen asentamista testaajan koneelle. MEKin ympäristön tarpeita varten SOAP- ja WCF-testiaskeliin on tehtävä kuitenkin seuraavat omat laajennukset:

[17] http://geekswithblogs.net/michaelstephenson/archive/2008/02/03/119235.aspx

- WCF-testiaskel, joka luo proxy-luokan suorituksen aikana alkuperäisen SOAP-testiaskeleen tapaan. Nopeuttaa normaalien www-sovelluspalveluiden testaamista.
- WCF-testiaskeleisiin tuki lähetysparametrien (client endpoint) konfiguroinnille testiaskeleessa erillisen konfiguraatiotiedoston sijaan.
- WCF-testiaskeleiden konfigurointimahdollisuus Arekin tarvitsemille WS-Security -laajennuksille. Vaihtoehtoisesti voidaan toteuttaa myös erillisenä testiaskeleena. Nämä voidaan toteuttaa myös SOAP-testiaskeleisiin.
- Testiaskeleet FTP-protokollalla tapahtuviin tiedostosiirtoihin Arek-eräajorajapintojen testausta varten. Tarvitaan toiminnot tiedoston lähettämiseksi ja noutamiseksi.

6.2.2 Hakemus- ja hoitojärjestelmän käyttöliittymätestaus

Käyttöliittymätestien automatisointi on MEK-projektissa vaikea kysymys. Järjestelmätestejä on hakemusjärjestelmän puolella paljon ja ne ovat testaajille erittäin työläitä, joten käsin suoritettuna ylläpitovaiheessa ei ole mahdollisuutta laajaan regressiotestaamiseen. Testausta vaikeuttaa myös CRM:n tapa käyttää lomakkeillaan JavaScriptiä tietojen tarkastamiseen, ja myös osa räätälöidyistä lomakkeista käyttää samaa tapaa. Näiden osien testaaminen ei käytännössä onnistu kuin käyttöliittymän kautta suoritettavilla testeillä. Tämän vuoksi käyttöliittymätestausta on pidettävä tärkeänä osana palvelupyyntöjen regressiotestausta, ja siksi työn aikana päädyttiin selvittämään tarkemmin myös sen automatisoinnin mahdollisuuksia.

Automatisoinnin esteenä on aiemmin ollut käyttöliittymän ja sen sisältämien toiminnallisuuksien epävakaus iteratiivisen kehitysmallin sekä ulkoisten häiriöiden vuoksi. Luvussa 3.10 mainitulla tavalla (Fewster & Graham 1999) tämä on yksistäänkin riittävä syy sille, ettei testejä ole kannattanut automatisoida. Testitiimin resurssipulan vuoksi myöskään työkaluja ei ole ehditty evaluoimaan aiemmin.

Nyt kun käyttöliittymän rakenne on vakiintunut, on hakemusjärjestelmän kehitys jo turhan pitkällä automatisoinnin vaatiman työn perustelemiseksi, mikäli ei saada varmuutta, että näitä testejä voidaan käyttää kauan. Testit joudutaan kuitenkin toistamaan käytännössä aina, kun lomakkeiden käsittelemissä ulkoisten rajapintojen tiedoissa tapahtuu muutoksia. Luvuissa 3.4 ja 3.10 käsitellyistä syistä tässä projektissa olisi perusteita vähintään testien suorituksen automatisoinnille (Kaner et al. 2002) – etenkin kun osa toiminnallisuuksista on pakko testata käyttöliittymän kautta. Näissä tilanteissa Meszaros (2003) puolustaa voimakkaasti käyttöliittymätestausta ja sen automatisointia.

Toisaalta testien käyttöikä voi riippua CRM-ohjelmiston versiosta. CRM:n versio 5.0 julkaistaan vuoden 2010 jälkimmäisellä puoliskolla (MS 2010). Vaikka versiopäivityksen tekemisestä ei vielä ole suunnitelmia, se voi tulla ajankohtaiseksi muutaman vuoden sisällä. Tämä on huomioitava automatisointipäätöstä tehdessä, koska päivitys voi aiheuttaa suuriakin muutoksia vanhoihin testeihin, mikäli käyttöliittymän rakenne muuttuu oleellisesti.

Hoitojärjestelmä on vielä siinä vaiheessa, että testien automatisointi olisi helpommin perusteltavissa työmäärällisesti, mutta testitapaukset ovat toisaalta paljon yksinkertaisempia, jolloin testin suorituksessa säästetty aika ei ole läheskään niin suuri kuin hakemusjärjestelmän puolella. Hoitojärjestelmän kannalta kriittisempi testauskohde on liiketoimintalogiikka sekä rahaliikennettä

ohjaavat liittymät. Näiden testaaminen on mahdollista sekä yksikkötestien että edellisessä luvussa esitellyn testiohjelmiston avulla.

Työkalujen evaluointi

Käyttöliittymätestausta varten arvioitiin kolmea eri testityökalua: Visual Studio 2008 Test Edition, WatiN:n v2.0 RC1 laajennettuna WatiN Test Recorder 2.0 beta 1228:lla. Dynamics CRM:n ja asiakasympäristön vaatimuksista johtuen työkalun on tuettava vähintään Internet Explorer -selaimen versiota 7.

Visual Studion web test -tallennustyökalu ei toiminut lainkaan CRM-ohjelmiston kanssa. Tallennuksen toisto kaatui jo aloitussivun lataukseen. Syyksi tähän paljastui JavaScript-tuen puute, koska testit suoritetaan .NETin HTTP-kerroksessa selainobjektin sijaan (MSDN 2010a; MSDN 2010b). Tämä estää käytännössä Visual Studion käytön CRM:lle rakennettujen sovellusten testaamiseen.

WatiN:n sekä sen tallennusohjelma näytti toimivan CRM:n kanssa hyvin, koska se suorittaa testit suoraan selainobjektia käyttäen. Näin järjestelmä tukee myös lomakkeilla olevia toimintoja hyvin. WatiN-skriptit tukevat myös ruutukaappausten ottamista testin aikana, mikä on hyvä asia esimerkiksi virheitä selvitettäessä. Testien toistovaiheessa havaittiin kuitenkin, että suurin osa navigointitoiminnoista oli tallentunut virheellisesti.

CRM:n näyttö koostuu useista, osin sisäkkäisistä kehyksistä ja taulukoista. Tallennuksessa osa tapahtumista oli kohdistettu väärään osioon. Toinen ongelma oli valittujen navigointilinkkien ja listanäytön tietueiden valinnassa, sillä näiden kohdalla tallennin valitsi usein sisäkkäisistä elementeistä väärän. Esimerkiksi listoilta tallennettiin taulukon solun painallukset sinänsä oikein, mutta soluilla ei ole yksilöllistä tunnistetietoa, vaan haku pitäisi kohdistaa solun sisällä oleviin muihin elementteihin. Kolmantena ongelmana olivat järjestelmän avaamat uudet ikkunat esimerkiksi tietueen avaamisen yhteydessä. Tallennin ei osaa linkittää uutta ikkunaa oikein siinä tehtäviin tapahtumiin, vaan tämä kytkentä piti korjata talletettuihin skripteihin käsin. Lisäksi tallennin listaa kaikki käytettävät kontrollit heti luodun testiskriptin alkuun. Tämä aiheuttaa ongelmia toistovaiheessa, koska osa kontrolleista löytyy vasta skriptin edettyä oikeaan valikkoon tai uuteen ikkunaan. Itse tietojen syöttöön ja painikkeiden painamiseen liittyviä osia tallennuksista voi hyödyntää, kunhan kontrollien etsimisen toteuttaa itse.

Käsin kirjoitetulla WatiN-skriptillä testaus kuitenkin onnistui, tosin toistossa havaittiin ensimmäisillä ajoilla ongelmia, joiden syy jäi epäselväksi. Skriptien kirjoittaminen käsin on kuitenkin hidasta, koska edellä mainittujen syiden vuoksi tallennetusta skripistä ei ole juurikaan apua.

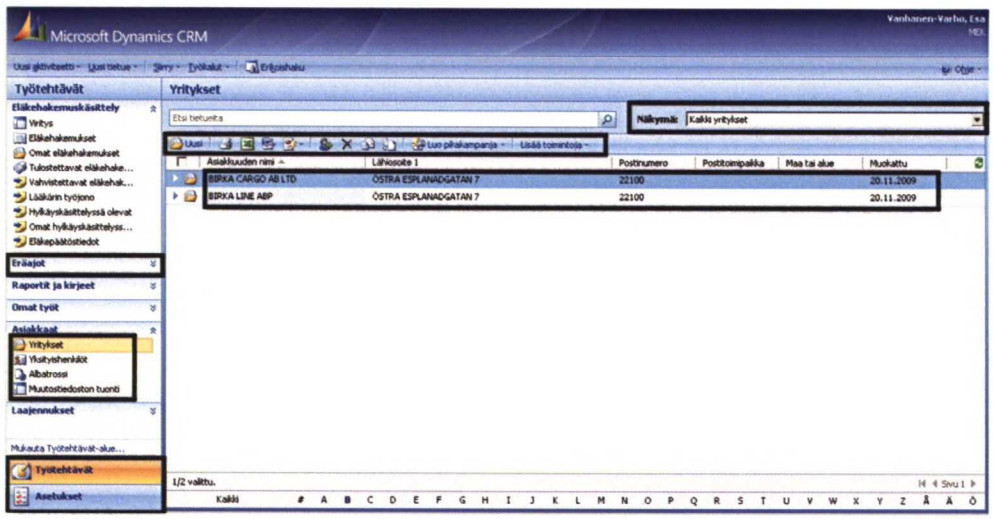
Yksinkertaisempien selainsovelluksien testaamiseen verkkopalveluiden puolella olisi mahdollista käyttää myös Visual Studion testejä. Näiden sovellusten kohdalla uudelleentestaamisen tarve on kuitenkin paljon pienempi, eikä testien yksinkertaisuuskaan puolla korkeita automatisointikustannuksia.

Tarvittavat laajennukset

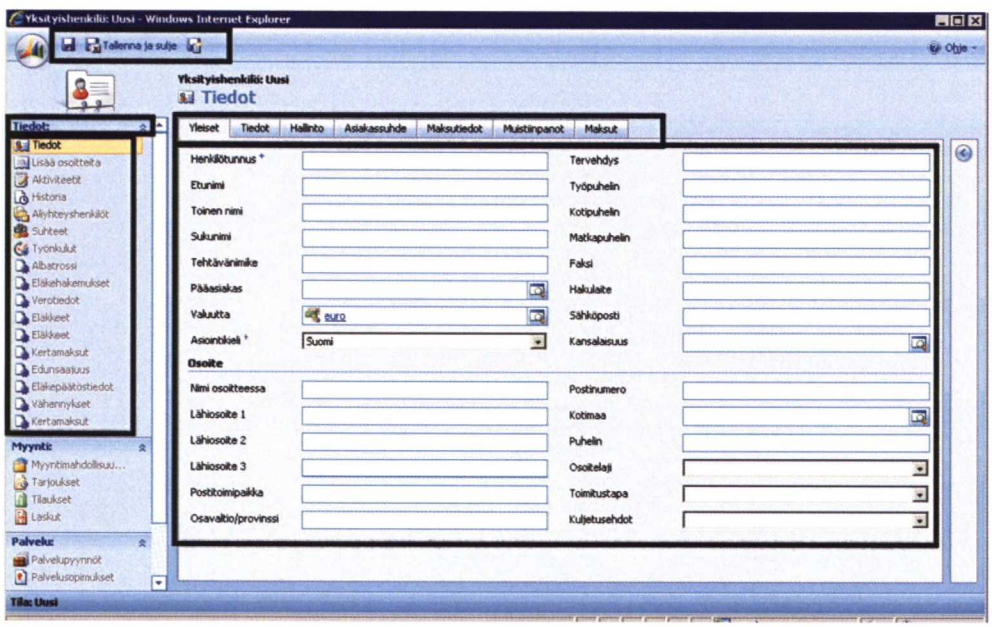
Käytännössä CRM:n käyttöliittymätestausta varten pitäisi toteuttaa oma testikirjasto, jolla halutut navigointikontrollit ja tietokentät löytyisivät helpommin. Oman kirjaston avulla olisi mahdollista toteuttaa avainsanaohjattu testiohjelma, jolloin tallentimen käyttöä ei edes tarvita. Jotta skriptien kirjoittaminen olisi testaajalle helppoa, tulisi kirjaston tukea kuvissa 6.3 ja 6.4 esitettyjen navigaatioja lomakekontrollien löytämistä joko näytöllä näkyvien tekstien tai lomakkeen sisältämien tietueiden teknisten nimien avulla. Osa syöttölomakkeista on CRM:n

ulkopuolisia laajennuksia, käytännössä erillisiä ASP.NET-sovelluksia, jotka on upotettu CRM:n vakiolomakkeelle IFRAME-kehyksiä käyttäen. Näille lomakkeille kontrollien hakujen pitää toimia teknisesti eri tavoin, mutta peittää tekniset erot testiskripteiltä.

Oman testikirjaston toteuttamiseen lupaavin lähestymistapa olisi BizUnit-ohjelmiston käyttö. Sen päälle on toteutettu jo nyt Microsoftin UI.Automation-luokkia käyttävä testikirjasto. Avainsanapohjaisuus ja Excel-tuki olisivat testaajien mielestä heille sopivia työkaluja, mikäli automatisointiin lähdetään.



Kuva 6.3: CRM:n päänäytön tärkeimmät kontrolliryhmät



Kuva 6.4: CRM:n tietuelomakkeen tärkeimmät kontrolliryhmät

Etenemistapa

Testikirjaston toteutus olisi työmäärältään suurehko. Ennen kirjaston toteutusta on siitä tehtävä tarkempi suunnitelma ja työmääräarvio. Realistisen arvion saamiseksi tulisi suorittaa ensin hieman laajempi konseptitestaus WatiN-työkalulle. Tämä

kannattaisi jakaa kahteen vaiheeseen toteuttamalla ensin yksinkertainen savutesti hoitojärjestelmän puolelta ja sen jälkeen hakemusjärjestelmästä yksinkertaisin mahdollinen sekä CRM:n sisältämiä vakiolomakkeita että räätälöityjä laajennuksia käyttävä testitapaus. Näin mahdolliset tekniset haasteet tulisi kartoitettua hyvin ennen lopullista toteutusta niin WatiN:n teknisen soveltuvuuden kuin toistoissa havaittujen suoritusongelmien osalta.

Tarkemman työmääräarvion lisäksi ennen lopullista päätöstä testikirjaston toteuttamisesta tulisi laatia riskianalyysi, jossa arvioidaan CRM:n versiopäivityksen vaikutuksia testikirjastolle ja -skripteille. Tässä suositeltava lähestymistapa olisi hankkia CRM 5.0:n ennakkoversio, jonka avulla käyttöliittymän mahdollisten muutosten vaikutukset voidaan kokeellisesti testata ja arvioida tarkemmin tarvittavien muutostöiden määrä.

6.3 Ulkoisten häiriöiden vaikutusten pienentäminen

Projektin aikana Arekin testiympäristöjen toistuvat katkokset ovat aiheuttaneet viivästyksiä kehitys- ja testaustyölle. Pelkästään suunnitelmallisia katkoksia on ollut paljon johtuen Arekin ympäristössä tapahtuvista muutoksista. Tämän lisäksi testihenkilöiden tiedot muuttuvat testauksen tuloksena, jolloin väärän henkilön käyttö testitapauksessa voi sekoittaa toisen testitapauksen lähtötiedot. Tämä on yhteneväistä Bruno et al. (2005) havaintojen kanssa (ks. luku 4.3). Testihenkilöiden tiedot voivat muuttua myös Arekin ympäristöpäivitysten yhteydessä, eikä projektiryhmä ole aina tietoinen näistä muutoksista.

Katkosten vaikutusten välttämiseksi ulkoiset liittymät voitiin tiettyyn pisteeseen asti korvata yksinkertaisella vakiotietoa palauttavalla tyngällä, mutta laajemman testauksen kannalta tyngän tulisi olla älykkäämpi – sen tulisi kyetä reagoimaan pyyntösanomaan, palauttamaan sen perusteella erilaisia vastauksia laajemmasta joukosta ja mahdollisesti jopa muokkaamaan vastauksen osia. Tällaisen tyngän käyttö mahdollistaisi myös vakioidun tiedon käyttämisen ilman testihenkilön tietojen muuttumista Arekin järjestelmissä, mikä olisi hyödyllistä virheitä korjattaessa, kun ajoja joudutaan toistamaan useita kertoja peräkkäin.

Sanomien monimutkaisuuden vuoksi niiden muodostaminen käsin ei ole mahdollista. Aineistojen luominenkaan ei ole realistinen vaihtoehto, koska aineiston sisällössä on paljon loogisia riippuvuuksia. Näin ollen työn aikana nousi esiin mahdollisuus tallentaa sanomaliikennettä ja käyttää tallennettua tietoa testauksessa. Testikäyttöä varten aineistoja on mahdollista nauhoittaa myös tuotantoaineistoista, kunhan niiden tunnistetietoja sekoitetaan ennen testipuolelle siirtämistä. Sekoittamisella tarkoitetaan esimerkiksi nimen ja henkilötunnuksen vaihtamista.

Tallennus-toistomallin käyttämistä palveluiden jäljittelemiseen ehdottivat luvussa 4.3.1 mainitusti Canfora ja Di Penta (2006). Projektissa oli aiemminkin käytetty tynkiä palauttamaan vakiovastauksia integraatiopalveluista kehitysvaiheessa. Sen lisäksi erään verkkopalvelun käyttämän rajapinnan virhetilanteiden selvityksen takia oli soapUI:lle laadittu sanoman sisällön perusteella vastauksen valitseva tynkä. Tämä havaittiin käytännössä hyvin toimivaksi ratkaisuksi, mutta soapUI:n WCF-tuen puute estää sen käytön laajentamisen. Näin ollen ratkaisua lähdettiin etsimään muista .NET-alustalla toimivista ilmaistyökaluista. Vaihtoehtoja löytyi käytännössä vain yksi, koska työkalun tulee olla muokattavissa MEK-projektin tarpeisiin sopivaksi.

6.3.1 Työkaluarviointi: MockingBird

MockingBird[18] on www-sovelluspalveluiden jäljittelyyn tarkoitettu avoimen lähdekoodin ohjelma .NET-ympäristöön. Se tukee usean palvelun samanaikaista jäljittelyä tiedostoihin talletettujen vastaussanomien avulla. Vastaussanomaksi palautetaan palvelukohtaisesti joko vakiovastaus tai pyyntösanoman sisällön perusteella valittu vastaus. Sanomien talletus tiedostoon on käytännöllistä, koska tällöin testaajat pääsevät muokkaamaan niitä tarpeen vaatiessa ilman erikois-työkaluja.

Ohjelman nykyinen versio 2.0RC vaatii jokaiselle erilliselle vastaussanomalle oman konfiguraatiorivin, josta esimerkki on kuvassa 6.5. Harmaat rivit ovat vastaavanlaisia konfiguraatiorivejä kuin näkyväkin. Tämä malli on hankala, jos sanomia halutaan tallentaa suuria määriä. Se myös hidastaa palvelun toimintaa, koska sisältökyselyjä suoritetaan rivi kerrallaan, kunnes ensimmäinen ehdon täyttävä rivi löytyy. Tämä ongelma voidaan ratkaista seuraavasti:

- Tallennetut sanomat nimetään palvelukohtaisesti vakioidulla tavalla siten, että tunnistamisessa käytetyt avaintiedot löytyvät tiedoston nimestä. Esim. `Vastaus_Palvelu1_Tunniste.xml`
- MockingBirdiin toteutetaan uusi ResponseStrategy-tyyppi `XPathExpression`. Tämä käsittely etsii XPath-kyselyn perusteella tunnistetiedon pyyntösanomasta alkuperäisen kuvassa 6.5 näkyvän käsittelyn tapaan, muttei tutki `Result`-elementin sisältöä vaan korvaa `FileName`-attribuutissa olevan`%Result%`-merkkijonon kyselyn tuloksella. Uudesta konfiguraatiotavasta on esimerkki kuvassa 6.6.

```
<Operation Name="GetPerson" ResponseStrategy="XPath">
  <XPathBasedResponses>
    <XPathBasedResponse>
      <Expression>/*[local-name()='Body']/*[local-name()='GetPersonRequest']/*[local-name()='Person']/*[local-name()='Name']</Expression>>
      <Result>James</Result>>
      <Response ResponseBehaviour="Standard" FileName="GetPersonJamesResponse.xml" ErrorType="None"/>
    </XPathBasedResponse>
    <XPathBasedResponse>...</XPathBasedResponse>
    <XPathBasedResponse>...</XPathBasedResponse>
    <XPathBasedResponse>...</XPathBasedResponse>
  </XPathBasedResponses>
</Operation>
```

Kuva 6.5: MockingBirdin alkuperäinen vastauskonfiguraatio

```
<Operation Name="GetPerson" ResponseStrategy="XPathExpression">
  <XPathBasedResponses>
    <XPathBasedResponse>
      <Expression>/*[local-name()='Body']/*[local-name()='GetPersonRequest']/*[local-name()='Person']/*[local-name()='Name']</Expression>
      <Response ResponseBehaviour="Standard" FileName="GetPerson%Result%Response.xml" ErrorType="None"/>
    </XPathBasedResponse>
  </XPathBasedResponses>
</Operation>
```

Kuva 6.6: MockingBirdin laajennettu yhden säännön vastauskonfiguraatio

6.3.2 Sanomien tallennus toistoa varten

Sanomien tallennukseen on mahdollista käyttää useita menetelmiä, kunhan huolehditaan tiedostojen nimeämisestä liittymäkohtaisesti sovitulla tavalla. Nimeämisstandardi on oleellinen, jotta aineistoja voidaan hyödyntää helposti suunnitelluilla MockingBirdin lisätoiminnallisuuksilla.

Liittymien toteutustavoista johtuen sanomia voidaan tallentaa ainakin BizTalk Serverin arkistotietokannasta joko hallintatyökalujen avulla tai erikseen toteutettavan sanomatietokantaa lukevan ohjelman avulla. Sanomaliikennettä

[18] http://mockingbird.codeplex.com/

tallennetaan nykyisin virhetilanteiden selvittämisen yhteydessä myös Wireshark-ohjelmalla[19] sekä ohjaamalla kyselyt Fiddler-välityspalvelinohjelmalle[20]. Fiddlerin käytön etuna on mahdollisuus tallentaa ohjelmallisesti kaikki halutut kyselyt ja niihin liittyvät vastaukset. Tiedostojen nimeämisessä voidaan tällöin käyttää pyynnön tai vastauksen sisältöä säännöllisten lausekkeiden avulla. WCF-rajapintojen kohdalla on mahdollista kytkeä päälle sanomien seuranta palveluiden konfiguraatiossa, mutta näiden lokitiedostojen hyödyntäminen vaatii jatkokäsittelyä, jotta varsinaiset sanomat saadaan irrotettua muista tapahtumatiedoista. Tämäkin on mahdollinen lähestymistapa liiketoimintakerroksen aineistojen keräämiseksi, ja lokitiedostojen käsittelyä varten on mahdollista laatia yksinkertainen ohjelma.

6.4 Ulkoisten liittymämuutosten vaikutusten tunnistaminen

Arek-liittymien suuren määrän ja liittymärajapinnoissa käytettävien XML-skeemojen monimutkaisuuden vuoksi Arekin versiopäivitysten muutosvaikutusten analysointi muodostuu hakemus- ja hoitojärjestelmän käyttöönoton myötä varsin työlääksi. Kaikki muutokset eivät välttämättä vaikuta MEKin käyttämiin toiminnallisuuksiin, ja näissä tapauksissa skeemojen tekninen päivittäminen integraatiopalveluihin ei ole aina välttämätöntä. Tällöin palvelun toimivuus voidaan varmistaa hyvissä ajoin Arekin testiympäristöissä suorittamalla olemassa oleva regressiotestipaketti luvussa 6.2.1 kuvatulla testaustyökalulla. Jos integraatiopalvelu taas vaatii päivitystä, sen toteutus olisi hyvä saada liikkeelle mahdollisimman nopeasti, jotta muutokset ehditään testaamaan Arekin yleensä varsin lyhyen hyväksymistestausjakson yhteydessä.

Skeemojen analysointi manuaalisesti esimerkiksi WinMerge- tai CsDiff-vertailuohjelmaa käyttäen mahdollistaa useista skeemoista koostuvan sanomakuvauksen muuttuneiden skeematiedostojen löytämisen nopeasti, koska työkalut osaavat vertailla muutoksia myös hakemistorakenteessa ja niiden sisältämissä tiedostoissa. Varsinaiset muutoskohdatkin löytyvät skeemoista helposti, mutta muutosten vaikutusten arviointi on jo monimutkaisempi työvaihe. Jos jossain skeemassa määritelty tietotyyppi muuttuu, sen vaikutus kohdistuu kaikkiin paikkoihin, missä tätä tyyppiä käytetään. Käytännössä nämä kaikki kohdat pitäisi etsiä jokaisesta erillisestä skeemasta. Sen jälkeen pitäisi vielä tutkia, esiintyvätkö muuttuneet kohdat MEKin sanomilla, jotta voidaan päätellä, tarvitseeko rajapintaa päivittää. Kuormittavuuden lisäksi nämä työvaiheet ovat myös virheherkkiä.

Liittymämuutosten vaikutusten tunnistamista olisi mahdollista helpottaa vertailutyökalun avulla. Työkalun tulisi sisältää kaksi toimintoa. Ensiksi työkalun tulisi tunnistaa, mitkä rakenteet XML-skeemassa ovat muuttuneet joko suoraan tai epäsuoraan tyyppimuutosten kautta. Toiseksi sen pitäisi osata tunnistaa, ovatko nämä rakenteet käytössä MEKin sanomilla. Sanomia voidaan kerätä talteen vertailua varten, kuten luvussa 6.3.2 kerrottiin.

Analysoinnin nopeuttaminen ja parempi varmuus muutosten vaikutuksista mahdollistaisivat ylläpitovaiheessa turhan päivitystyön välttämisen. Lisäksi parempi tieto muutoksen vaikutuksista ohjaisi keskittymään regressiotesteissä muutoskohtien tarkempaan testaamiseen.

[19] http://www.wireshark.org/
[20] http://www.fiddler2.com/fiddler2/

Tämä lähestymistapa ei välttämättä toimi kahden monimutkaisimman aineiston, (työsuhdeote ja eläkkeen laskentapalvelun vastaus) kohdalla, koska niistä ei ole vielä saatavilla www-sovelluspalvelurajapintaa ja sen mukaista WSDL-kuvausta tai sanomien XML-skeemoja. Palveluista on tulossa XML-pohjaiset versiot tulevaisuudessa, mikä helpottaa jatkossa näidenkin sanomien rakenteiden vertailua. Työkalua voi olla mahdollista soveltaa Word-dokumenteista muunnettuihin omiin BizTalk-sanomakuvauksiin, mutta tämän prosessin virheherkkyydestä johtuen työkalun käyttökelpoisuus on varmistettava myöhemmin erikseen.

6.4.1 Tutkimustietoa skeemamuutosten tunnistamisesta

XML-skeemamuutosten vaikutusten tunnistamiseen liittyvää tutkimusta ei juuri löytynyt. Guerrini et al. (2005) sekä Passi et al. (2009) ehdottivat muutosten kuvaamista perustoimintojen avulla. Näitä ovat esimerkiksi attribuuttien tai elementtien lisäys ja poisto. Tämä lähestymistapa edellyttää skeeman muuttamista näitä perustoimintoja käyttäen ja muutosten kirjaamista. Menetelmät soveltuvatkin näin ollen skeemaa kehittävän osapuolen käyttöön, joten niistä ei ole apua skeemoja käyttävälle osapuolelle. Leonardi et al. (2007) ovat tutkineet DTD-kuvausten muutosten tunnistamista, mutta menetelmän soveltaminen XML-skeemoihin on vasta suunnitteilla.

Coatesin ja Duin (2010) esittivät idean "laajennetuista XPath-poluista". Heidän menetelmänsä tiivistää skeeman mahdolliset sanomarakenteet kuvassa 6.7 nähtävään esitysmuotoon, josta ilmenevät myös mahdolliset tietotyyppien muutokset. Näitä muutoksia tekijät kutsuvat "piilotetuiksi muutoksiksi", koska tyyppimuutos voi vaikuttaa useampaan paikkaan varsinaisessa sanomassa. Tässä esitysmuodossa tyyppimuutosten vaikutus puretaan suoraan kaikkiin niihin rakenteisiin, jossa muuttunutta tyyppiä käytetään. Idea tälle tutkimukselle oli noussut käytännön projekteissa havaituista muutosten arvioimisen haasteista, aivan kuten MEK-projektissakin. Tekijät ovat laatineet ideastaan myös Apache-lisenssin alla levitettävän XMLZebra-ohjelman[21].

```
/era:MELIlmoitusPyyntoEra
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:EranId[base=xs:decimal]?
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:EranId[base=xs:decimal]?/xs:fractionDigits/@value=0
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:EranId[base=xs:decimal]?/xs:minInclusive/@value=-2147483648
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:EranId[base=xs:decimal]?/xs:maxInclusive/@value=2147483647
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Palvelutunnus[base=xs:string]
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Palvelutunnus[base=xs:string]/xs:minLength/@value=5
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Palvelutunnus[base=xs:string]/xs:maxLength/@value=12
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Aikaleima[base=xs:dateTime]
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Lahettaja[base=xs:string]
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Lahettaja[base=xs:string]/xs:minLength/@value=2
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Lahettaja[base=xs:string]/xs:maxLength/@value=5
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Kasittelija[base=xs:string]?
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Kasittelija[base=xs:string]?/xs:minLength/@value=0
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:Kasittelija[base=xs:string]?/xs:maxLength/@value=64
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:EranTunnus[base=xs:string]?
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:EranTunnus[base=xs:string]?/xs:minLength/@value=0
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:EranTunnus[base=xs:string]?/xs:maxLength/@value=10
/era:MELIlmoitusPyyntoEra/eraYleiset:EranTiedot/eraYleiset:TyonantajanTunniste[base=xs:string]?
```

Kuva 6.7: XMLZebra-ohjelman tuottama esitysmuoto skeeman rakenteesta

6.4.2 Ratkaisumalli

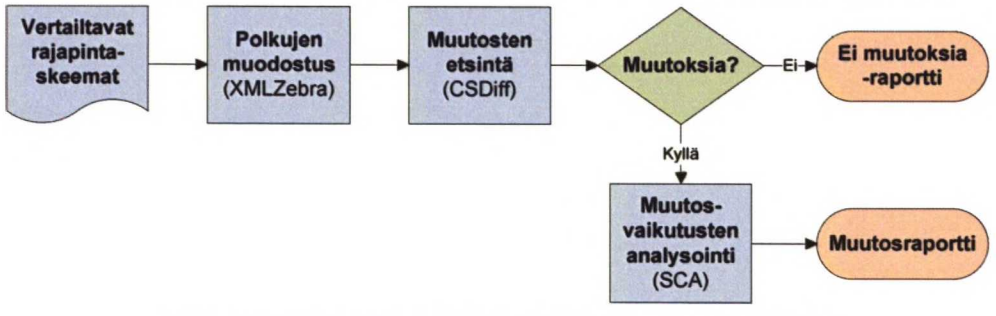
Liittymämuutosten tunnistamiseen syntyi kuvassa 6.8 esitetty prosessi, joka hyödyntää vapaasti saatavilla olevia ohjelmia oman vertailukomponentin lisäksi. Rajapintaskeemojen latausta lukuun ottamatta prosessi voidaan automatisoida komentorivisarjoilla suoritettaviksi.

Ensimmäisessä vaiheessa vertailtaville skeemoille ajetaan XMLZebra-ohjelmalla laajennetut XPath-esitysmuodot rajoittaen sanomarakenteet varsinaisen

[21] https://launchpad.net/xmlzebra

palvelusanoman juurielementistä lähteviksi. Skeemaversioiden eroja voidaan tutkia kuvassa 6.7 esitetystä muodosta varsin helposti tekstivertailuohjelmaa käyttäen, mihin tässä ratkaisussa käytetään CSDiff-ohjelmaa[22] komentoriviltä käynnistettynä. Sen tuloksena saadaan diff-formaattia noudattava vertailutiedosto. Jo tämän esitys-muodon avulla muutosten vaikutukset ovat tulkittavissa suoraan. Esitystavasta voidaan myös laatia käsin XPath-kyselyt, joilla muuttuneita rakenteita voi hakea malliaineistoista.

Jatkokäsittelyä varten laaditaan SchemaChangeAnalyzer-työkalu (SCA), joka muodostaa diff-tiedoston perusteella nämä muutoksiin kohdistuvat XPath-kyselyt ja suorittaa ne hakemistoon tallennettuihin mallisanomiin. Mikäli muuttuneita tai poistuneita rakenteita esiintyy suoraan mallisanomilla, ohjelma raportoi muutokset päivitystä vaativiksi. Jos rakenteita ei esiinny, muutos luokitellaan todennäköisesti vaikutuksettomaksi. Lisätyt rakenteet raportoidaan erikseen. Niiden kohdalla tutkitaan, kohdistuuko elementin tai attribuutin muutos suoraan johonkin käytössä olevaan tietorakenteeseen tai sen alle, ja onko lisäyksen esiintyminen sanomalla pakollista vai vapaaehtoista. Näistä tekijöistä riippuen muutoksen aiheuttamista vaikutuksista annetaan eritasoisia ilmoituksia. Käyttäjää kehotetaan kuitenkin tutkimaan diff-tiedostosta ilmenevät muutokset myös itse.



Kuva 6.8: Liittymämuutosten vaikutusten tunnistaminen

6.5 Muut muutokset testaus- ja kehitysprosessiin

6.5.1 Testattavuus

Testattavuuden parantaminen etenkin hakemusjärjestelmän osalta havaittiin tarpeelliseksi ennen ylläpitovaiheeseen siirtymistä (MEKAUD 2010). Hakemusjärjestelmän koodi ei noudattanut kaikilta osiltaan monikerrosarkkitehtuuria. Hakemusjärjestelmän toteutuksen alkaessa siirryttiin iteratiiviseen kehitysmalliin, ja sen alkuvaiheessa vallinneen resurssipulan vuoksi refaktoroinnille ei jäänyt riittävästi aikaa. Tämä on mainittu iteratiivisen mallin haasteeksi (Poimala et al. 2008).

Koodin refaktorointia on tehty jo tämän työn aikana. Muutosten yhteydessä pyritään kunkin luokan tilannetta parantamaan poistamalla arkkitehtuurikerrosten ohitukset ja siirtämään toiminnallisuuksia oikealle tasolle. Yksikkötestauksen helpottamiseksi poistetaan myös luokkariippuvuuksia. Vastaavaa refaktorointia ja yksikkötestattavuuden parantamista tehdään myös integraatiopalveluiden liiketoimintalogiikkaa suorittaviin luokkiin muutosten yhteydessä.

[22] http://www.componentsoftware.com/products/CSDiff/index.htm

Testattavuuden parantaminen on mainittu edellytyksenä testiautomaation onnistuneelle toteuttamiselle (Berner et al. 2005; Kaner et al. 2002; Bach 1999). Tätä käsiteltiin luvuissa 3.1 ja 3.4.

6.5.2 Testauksen kattavuus ja mittaukset

Yksikkötestausta tulisi laajentaa nykytasosta, jotta ylläpitovaiheessa olisi käytössä riittävän laajat regressiotestipaketit. Yksikkötestaus on kuitenkin helpoin automatisoitava testaustaso, ja sen avulla voidaan ehkäistä turhien regressiovirheiden syntyminen (ks. luvut 3.2, 3.4 ja 3.10). Vaikeasti testattavissa komponenteissa luokkatason yksikkötestaus ei ole tärkein kriteeri (Kaner et al. 2002; ks. luku 3.4). Näiden kohdalla testaus voidaan toteuttaa myös integraatiotestauksena alustamalla testin alkutilanne tietokantaan ja käyttämällä oikeita mallisanomia tyngissä. Testauksen laadun parantaminen on Bachin (1999) sekä Kaner et al. (2002) mukaan tärkeämpi tekijä ohjelmiston laadulle kuin testien automatisointi, joten ilman automatisointiakin testien kattavuuden parantaminen on perusteltua.

Projektin testauskriteereistä puuttuu selvä määritelmä testauksen kattavuusvaatimuksista lukuun ottamatta vaatimusten toiminnallista testaamista. Jälkikäteen kattavuusvaatimuksen asettaminen ei käytännössä onnistu ilman uudelleenresursointia ja lisätyötä. Uusissa hankkeissa tulisikin määritellä selvä yksikkötestauksen kattavuustavoite, jotta testit ovat varmasti riittävän kattavia ylläpitoa ajatellen. Näin kehittäjillä olisi tieto vaaditusta testaustasosta ja toisaalta yksikkötestaukseen tarvittavat resurssit on mahdollista arvioida realistisesti. Kattavuustavoitteen asettamista suosittelevat Burnstein (2003) ja IEEE:n yksikkötestausta koskeva standardi (Std 1008-1987), joita käsiteltiin luvuissa 2.3.2, 2.3.3 ja 2.4.2.

Yksikkötestauksen kattavuutta olisi mahdollista parantaa käyttämällä rakenteellisten testitapausten luomiseen Microsoftin Pex-työkalua (ks. luku 3.9). Samassa paketissa on mukana Moles-laajennus kooditason rajapintojen jäljittelyyn. Mikäli Microsoft jatkaa Molesin kehitystä, kannattaa nämä työkalut ottaa käyttöön viimeistään Visual Studio 2010 käyttöönoton yhteydessä. Perusteluna tälle on projektin tavoite käyttää mahdollisimman paljon kehitystyökalujen sisältämiä vakiokomponentteja tai Microsoftin tukemia omia laajennuksia erillisten työkalujen sijaan.

Integraatiopalveluiden testausta voidaan parantaa BizUnitin käyttöä laajentamalla. Luvussa 6.2.1 mainittujen ominaisuuksien lisäksi se tukee suoraan erilaisia integraatiopalvelun käyttämiä rajapintoja, kuten tiedostojen kopiointia haluttuihin hakemistoihin ja MSMQ-sanomajonon käyttöä. Uudessa versiossa on tuki myös aiemmin vaikeasti testattavien muunnosten ja vastaanotto/lähetyskäsittelyjen (ns. pipeline) yksikkötestaukselle. BizUnit-testien laatimiseen tarkoitettua graafista BizUnit Designer[23] -ilmaistyökalua ei voida ottaa käyttöön, koska sen kehitys näyttää keskeytyneen, eikä se tue enää uusimpia BizUnit-versioita. BizUnitin käyttöönoton yhteydessä on mahdollista arvioida tarkemmin myös Gallio-testiautomaatioalustan[24] käyttökelpoisuutta kokeilemalla testien suorittamista sen kautta.

Testien laatua ja kattavuutta voidaan parantaa myös käyttämällä toiminnallisten testitapausten laatimiseen luvussa 3.9 mainittua ACTSia. Sen avulla voidaan luoda nopeasti laajoja testijoukkoja, jotka voidaan suorittaa tietopohjaisen

[23] http://bud.codeplex.com/
[24] http://www.gallio.org/

testiajurin läpi. ACTSin käyttöä pilotoidaan BizUnitin käyttöönoton yhteydessä. Tätä varten laaditaan muunnostyökalu, joka sijoittaa ACTSilla luodut arvot XML-muotoisten testiaineistojen sisään haluttuihin paikkoihin. Käytännössä tämä tapahtuu erillisellä konfiguraatiotiedostolla, jossa kutakin ACTS-taulukon saraketta kohden voi olla 0...n XPath-lauseketta, joiden osoittamiin paikkoihin arvo sijoitetaan.

XML-aineistojen luomiseksi TAXI-työkalun (Bertolino et al. 2007) kokeileminen vaikuttaisi mielenkiintoiselta lähestymistavalta, mutta sen lisenssiehtoja ei kerrota selvästi verkkosivuilla, joten nämä pitäisi selvittää ensin. Www-sovelluspalvelujen laajempaan testaamiseen tarkoitetun WS-TAXI-työkalun (ks. luku 4.3) kehitystä kannattaa seurata, sillä tekijät suunnittelevat sen julkaisua verkosta ladattavaksi.

6.5.3 Elinkaaren hallinta

Regressiotestitapausten valinta- ja priorisointimenetelmän mahdollista käyttöönottoa selvittäessä ongelmaksi nousi tiedon puute vaatimusten, vikojen ja testitapauksien sekä ohjelmakoodin muutosten yhteyksistä toisiinsa eri ympäristöissä sijaitsevien Rational ClearQuestin ja Team Foundation Serverin välillä. Integraatio tai tietojen siirto TFS:ään todettiin liian raskaaksi ja riskialttiiksi operaatioksi kehitysprojektin aikana.

Kahden järjestelmän käyttö projektissa sai alkunsa, kun projektin varhaisessa vaiheessa tehtiin kevyitä mallitoteutuksia siitä, mitä Microsoft-arkkitehtuuriin siirtyminen voisi käytännössä tarkoittaa. Päätoimittajan toimitusmenetelmään kuuluvien Rational-järjestelmien valinta oli tuolloin loogista. Iteratiiviseen kehitysmalliin siirtymisen jälkeen testattavia versioita julkaistiin selvästi aiempaa useammin ja testauksen kohteena oleva hakemusjärjestelmä oli laajempi kuin aiemmat sovellukset. Vasta tässä yhteydessä havaittiin tarve vaatimusten ja testaustietojen yhdistämiselle versionhallinnassa olevaan ohjelmakoodiin.

Tähän ongelmaan ei esitetä suoraa ratkaisua tässä vaiheessa projektia, mutta asia on tiedostettu tulevia hankkeita varten. Ennen ylläpitovaiheeseen siirtymistä migraatiomahdollisuus selvitetään uudelleen. Tietojen ylläpito kahdessa tai useammassa paikassa ei ole pienemmälle ylläpitotiimille järkevää. Vaihtoehtoisesti vanhoja tietoja ei siirretä, mutta jatkokehityksen hallinta siirretään kokonaan TFS:lle. Tämä keventää ylläpitoprosessia tarvittavien järjestelmien suhteen.

Samankaltaisissa yhteisprojekteissa pyritään jatkossa yhden järjestelmän käyttöön. Tämä edellyttää toimittajien menetelmien ja raportoinnin yhteensovittamista heti projektin alusta alkaen. Siten esimerkiksi vioista kertyy tarkempaa tietoa kooditasolla, jolloin moduulien virheherkkyyttä voidaan analysoida tarkemmin. Tämä tieto auttaisi keskittämään testaamista oikeisiin kohteisiin, koska virheillä on tapana kasautua (ks. luku 2.5.3). Yhden järjestelmän käyttö tuottaisi testaajille myös tarkempaa tietoa uusien versioiden sisältämistä muutoksista, mikä ohjaisi testaamista tarkemmin ja pienentää riskiä siitä, etteivät kehittäjät muista raportoida kaikkia muutoksia esimerkiksi refaktoroinnista testaajille.

6.5.4 Regressiotestien valinta ja priorisointi

Työn aikana pyrittiin löytämään tapoja regressiotestien valinnalle tai priorisoinnille, koska täydellistä uudelleentestausta ei voida ylläpitovaiheessa suorittaa jokaisen muutoksen yhteydessä. Ilman luvussa 6.5.3 käsiteltyä kehitys-

järjestelmien tietojen yhdistämistä ainoa käyttökelpoinen priorisointimenetelmä olisi Srikanth et al. (2005, ks. luku 2.5.3) esittämä vaatimuksiin perustuva menetelmä. Testaajat tekevät samankaltaista priorisointia epämuodollisesti jo nyt, joten tämän menetelmän käyttö toisi priorisointiin ainoastaan hieman lisää selkeyttä. Testaajat kaipaisivat testien priorisointipäätösten tueksi enemmän nimenomaan koodimuutoksiin ja virheiden sijaintiin liittyvää tietoa, jota ei siis nykyisin ole helposti saatavilla. Tämän vuoksi myöskään näitä tietoja hyödyntäviä menetelmiä ei voida ottaa käyttöön.

Jatkoprojekteissa kehitystyön hallinta yhdessä järjestelmässä antaisi paremmat edellytykset valinta- ja priorisointimenetelmien käytölle. Testien kattavuustietoihin perustuvien menetelmien käyttöönottoa kannattaa harkita vasta, kun tarvittava tieto on saatu keskitettyä.

Myös UML-pohjaisten menetelmien käyttöönottoa kannattaa harkita jatkohankkeissa, koska UML:ää käytetään muutenkin määrittely- ja suunnitteluvaiheissa dokumentointiin. Menetelmien käyttöönotto parantaisi tuotetun dokumentaation tarkkuutta, eikä niiden käyttäminen toisi suurta lisätyömäärää. Etenkin Chen et al. (2002, ks. luvut 2.5.2 ja 2.5.3) menetelmä auttaisi sekä testien valinnassa että priorisoinnissa.

6.6 Muutosten alustava priorisointi

Tässä luvussa esitetään alustava järjestys muutosten toteutukselle huomioiden niiden vaatima työmäärä ja hyödyllisyys jo käynnissä olevan toteutusvaiheen aikana ennen siirtymistä ylläpitoon.

Kehitys- ja testiympäristön ulkoisista rajapinnoista aiheutuvan epävakauden poistaminen MockingBird-alustan käyttöönotolla on helposti toteutettava tehtävä, joka voidaan aloittaa perustoiminnoilla jo ennen alustaan ehdotettujen laajennusten toteuttamista, kunhan testaajien kanssa saadaan kerättyä tarvittavat testitapaukset talteen. Konfiguroinnin helpottamiseksi tarvittavan laajennuksen pitäisi olla toteutukseltaan hyvin yksinkertainen, joten suurempienkaan testiaineistomäärien ylläpidolle ei pitäisi olla esteitä.

BizUnit-ohjelman käyttöönoton laajentaminen voidaan ajoittaa työnantajien kuukausi-ilmoitusliikennettä hoitavan Ankkuri-portaalin toukokuussa 2010 tapahtuvan huoltopäivityksen yhteyteen. Samalla BizUnit kannattaa ottaa eläkkeen hoitojärjestelmään liittyvien tiedostorajapintojen sekä tietokannan tilan testaustyökaluksi. Ankkuri-palvelun päivitysten yhteydessä voidaan rakentaa tarvittavat testitapaukset niin ulkoisten rajapintojen suoraa testausta kuin integraatiopalveluiden sisäistä toiminnallisuutta varten.

Ankkuri-palvelun sisäisiin rajapintoihin liittyy myös kokonaisprosessien testaus. Sanoman monivaiheisen käsittelyn toimivuus BizUnitissa voidaan varmentaa toteuttamalla testitapaus, jossa portaaliin lähetettävän ilmoituksen käsitteleminen vaatii erillisen vahvistusviestin lähettämistä ensimmäisen paluuviestin jälkeen. BizUnitin sisältämien tietokantaoperaatioiden sekä tiedostovertailujen toimivuus saadaan varmistettua eläkkeen hoitojärjestelmän maksutietokannan ja siitä tuotettujen maksatustiedostojen testien yhteydessä. Samalla selviää, tarvitaanko näihin toiminnallisuuksiin omia laajennuksia tai tarkempia virheraportteja. BizUnitin käyttöönoton ohella on mahdollista kokeilla pienimuotoisesti Gallio-testiautomaatioalustaa ja arvioida sen käytettävyyttä.

Ankkuri-päivitysten yhteydessä kokeillaan myös testitapausten luontia ACTS-työkalua käyttäen. Sen avulla saadaan luotua suuri määrä aineiston sisältö-

tarkastuksiin liittyviä testitapauksia, jotka yhdistelevät erilaisia sisältövirheitä käsin laadittuja testejä monipuolisemmin. Tämä mahdollistaa tarkastus-komponentin perusteellisemman testauksen, joka voi paljastaa mahdolliset puutteet usean samanaikaisen sisältövirheen raportoinnin toimivuudessa. Komponentin tulisi oikein toimiessaan raportoida kaikki virheet yhdellä käsittelyllä, jotta aineiston lähettäjä voi korjata koko aineiston kerralla oikean muotoiseksi. Skeemamuutosten tunnistustyökalun lopullinen toteutus ajoittunee Ankkuri-portaalin päivityksien jälkeiseen työvaiheeseen, jolloin Arekista on saatavissa uusia muutoksia sisältäviä rajapintakuvauksia.

Ohjelmistojen testattavuuden parantaminen on ollut käynnissä jo työn aikana, ja tätä jatketaan luonnollisesti edelleen. Testien kattavuuden parantamiseksi Pexin käyttöä pilotoidaan erikseen valitussa komponentissa, mikäli työkalu saadaan integroitua nykyiseen Visual Studio 2008 -ympäristöön. Muussa tapauksessa sen käyttöä lykätään kehitysympäristön versiopäivitykseen asti.

Yksikkötestien automatisointi kehitysympäristössä on myös käynnistetty työn aikana kehitystiimin toimesta. Automatisoinnin laajentamiseksi selvitetään myös mahdollisuudet erillisen automaattitestiympäristön pystyttämiseen. Ainakin tietokannat ja CRM-järjestelmä pitäisi pystyä eristämään manuaalisista testiympäristöistä, mikäli automatisointia tehdään paljon (ks. luku 3.4; Fewster & Graham 1999; Martin 2005).

Optimitilanteessa kehitysversio saataisiin julkaistua aina haluttaessa erilliseen testiympäristöön ja testattua siellä ilman ihmisen toimenpiteitä. Käytännössä joudutaan kuitenkin miettimään, mitkä vaiheet ovat oikeasti sellaisia, että automatisoinnista saadaan eniten kustannushyötyjä. Jo tässä vaiheessa on selvää, että kaikkia ehdotuksia ei ole realistista toteuttaa esitetyllä tavalla.

7 Johtopäätökset

Tässä luvussa arvioidaan työssä saavutettuja tuloksia sekä itse työtä. Luvun lopussa esitetään myös suunnitelmia jatkokehitykselle.

7.1 Tulokset

Tämän työn tuloksena syntyi suunnitelma Merimieseläkekassan järjestelmien ylläpidon ja regressiotestauksen tehostamiseksi. Osa suunnitelmasta koskee toimintatapojen muutoksia, osa työkalujen toteutusta ja käyttöönottoa. Projektin aikataulullisista tekijöistä johtuen suunnitelmaa ei ollut mahdollista toteuttaa ja mitata työn aikana. Näin ollen suunnitelman toimivuutta ja hyötyjä ei ole voitu vahvistaa kokeellisesti lopullisessa mittakaavassa. Projektin kannalta tämä ei ole ongelma, koska suurin osa työn kohteena olevista järjestelmistä siirtyy tuotantokäyttöön vasta vuoden 2010 lopussa. Laajemmat suunnitelman toteutustehtävät priorisoidaan niiden kehitystyötä helpottavien ominaisuuksien perusteella loppuvuoden työsuunnitelmiin.

Työssä tutkittiin yleisten periaatteiden soveltuvuutta tietyn asiakasprojektin tarpeisiin, joten tulokset eivät ole välttämättä sellaisenaan hyödynnettävissä muissa ympäristöissä. Valituille lähestymistavoille löytyi kuitenkin perusteita kirjallisuudesta.

Seuraavissa luvuissa käsitellään tarkemmin luvussa 1.2 esiin tuotuja työn keskeisimpiä tavoitteita.

7.1.1 Palvelupyyntöjen ja kokonaisprosessien testaus

Palvelupyyntöjen testauksen tehostamiseen liittyi työn tavoitteissa mainittu kysymys: "Kuinka palvelupyyntöjen ja niistä muodostuvien kokonaisprosessien testausta voidaan tehostaa? Minkälaisia testaustyökaluja tämä edellyttää?" Tätä pidettiin keskeisimpänä testaustyötä aiheuttavana tekijänä ulkoisten järjestelmien muutoksista ja rajapintojen lukumäärästä johtuen.

Alkuperäinen ajatus graafisen käyttöliittymän kautta toimivasta valmisohjelmistosta ei toteutunut, koska ohjelmat eivät sisältäneet riittävää toiminnallisuutta. Monipuolisimmista SOAP-protokollaa tukevista ohjelmistoista puuttui riittävä tuki Windows-ympäristössä tärkeille WCF-rajapinnoille. Asetuksia yksinkertaistamalla näitä ohjelmia olisi ollut mahdollista käyttää, mutta toisaalta se olisi rajannut jatkossa teknisiä toteutusmahdollisuuksia sisäisten järjestelmien osalta. Riittävää rajapintatukea tarjoavat testausohjelmat eivät puolestaan tukeneet kokonaisprosessien testaamista laajemmin ainakaan ilman merkittäviä itse toteutettuja testikehyksiä.

Tästä syystä, hieman yllättäenkin, integraatiopalveluissa käytössä olevan BizTalk Server -ohjelmiston yksikkö- ja järjestelmätestaukseen tarkoitettu BizUnit-ohjelmisto nousi uusimmilla versioillaan varteenotettavaksi alustaksi palvelupyyntöjen testaamiseen hyvän rajapintatukensa ja helpon laajennettavuutensa ansiosta. BizUnit täytti graafista käyttöliittymää lukuun ottamatta kaikki testauksen kannalta keskeiset tarpeet joko suoraan tai yksinkertaisilla omilla laajennuksilla. Avainsanapohjainen lähestymistapa ja valmis tuki taulukkolaskentapohjaiselle testitapausten esitystavalle tekevät ohjelmasta käyttöliittymän puutteesta huolimatta helposti lähestyttävän myös kehitystyötä tekemättömille testaajille.

BizUnitin käyttöönotto priorisoitiin heti aloitettavaksi toimenpiteeksi erään osajärjestelmän ylläpitotehtävien yhteydessä. Tässä yhteydessä toteutetaan myös projektiympäristön tarvitsemat rajapintalaajennukset. Näiden kokemuksien myötä voidaan arvioida paremmin työkalun hyötyjä testaamisen tehostamisessa.

Työn aikana tunnistettiin myös eläkehakemusjärjestelmän kohdalla tarve suorittaa osa palvelupyyntöjen testauksesta hakemusjärjestelmän käyttöliittymän kautta, koska käyttöliittymäkerros sisältää myös aineiston käsittelyyn liittyviä toiminnallisuuksia. Tätä asiaa käsitellään erikseen luvussa 6.2.2.

7.1.2 Regressiotestien määrän pienentäminen

Regressiotestien määrän pienentämiseen etsittiin menetelmiä sen jälkeen, kun työn aikana havaittiin hakemusprosessiin liittyvien testitapausten suuri määrä. Erilaisia valinta- ja priorisointimenetelmiä tutkittiin laajasti, mutta niiden käyttöönotolle ei löytynyt edellytyksiä.

Suurin syy menetelmien käyttökelvottomuudelle oli tiedon puute järjestelmissä. Vaatimusten, testien ja vikojen hallinta tapahtuu eri järjestelmässä kuin itse kehitystyö ja ohjelmakoodin versionhallinta. Käytännössä kaikki menetelmät vaativat ensin mainittujen yhdistämisen jälkimmäisiin. Suurin osa menetelmistä tarvitsee tuekseen myös testien kattavuusmittausten tietoja. Jälkikäteen näiden tietojen yhdistäminen ei enää onnistu järkevillä työmäärillä, ja yhdistämisen onnistuessakin kehitysprosessin aikaiset historiatiedot olisi jo menetetty. Näin ollen ei ollut perusteltua käyttää suuria työmääriä rajallisen hyödyn tavoitteluun vaan jatkaa testaajien osaamiseen ja vaatimusten priorisointiin perustuvalla nykyisellä epämuodollisemmalla testien priorisointimallilla.

Tiedon puute kehitysprosessin järjestelmissä on huomioitu tulevia projekteja ajatellen.

7.1.3 Palvelurajapintojen muutosten vaikutusten arviointi

Työn tavoitteissa esitettiin kysymys: "Kuinka palvelurajapintojen muutosten vaikutuksien arviointia voidaan helpottaa?" Tämä sinänsä pieneltä yksityiskohdalta vaikuttava seikka nousi merkittäväksi ulkoisten rajapintojen määrän sekä niiden monimutkaisuuden vuoksi. Tuhansia tietueita sisältävän sanomarakenteen analysointi olisi erittäin vaikeaa ihmissilmin tehtynä.

Luvussa 6.4.2 esitelty prosessi ja oma analyysityökalu vaikuttavat lupaavalta lähtökohdalta analysoinnin nopeuttamiseen. Työn tehostumista voidaan arvioida jonkin tulevan sanomarajapinnan muutoksen yhteydessä suorittamalla sama työ sekä aiemman manuaalisen toimintamallin että työkalun kautta.

7.1.4 Ulkoisten rajapintojen häiriöiden vaikutusten vähentäminen

Ulkoisten rajapintojen häiriöiden vaikutusten vähentämiseen etsittiin ratkaisua laatimalla malli tyngän avulla tapahtuvaan palveluiden jäljittelyyn. Tässä ratkaisutavassa ei sinänsä ollut mitään uutta, mutta tässä projektissa ongelmana oli rajapintojen monimutkaisuus ja lukumäärä. Jokaisen rajapinnan toteuttaminen erikseen ja niiden ylläpito muutostilanteissa vaatisi liikaa työtä.

Löydetty vapaaseen lähdekoodiin perustuva MockingBird-alusta tarjoaa helposti käyttöönotettavan työkalun palvelujen jäljittelylle, etenkin kun mallisanomien tallentaminen testien yhteydessä on suhteellisen helppo prosessi. Esitetty suoraan sanomasisältöön perustuvan konfiguraatiotavan lisääminen ohjelmaan

mahdollistaa sen helpon käyttöönoton myös tilanteissa, joissa erilaisia testiaineistoja tarvitaan suuria määriä. Järjestelmän laajennettavuus antaa myös mahdollisuuksia muihin itse laadittuihin ratkaisuihin myöhemmin.

7.1.5 Käyttöliittymätestaus

Työn aikana paljastui myös eläkehakemusjärjestelmään sisältyvien palvelukutsujen ja niiden muodostamien kokonaisprosessien kohdalla tarve suorittaa osa testeistä käyttöliittymäkerroksen kautta sen sisältämien toiminnallisuuksien vuoksi.

Tästä syystä suunniteltiin toteutustapa selainpohjaisesti toimivan Microsoftin CRM-järjestelmän sekä siihen itse laadittujen lisätoiminnallisuuksien testaamiseen. Suunniteltu malli erillisen WatiN-testikehyksen yhdistämisestä BizUnitin avainsanapohjaiseen ohjaukseen on mahdollinen toteuttaa, mutta yhdistämisen työmäärä on arvioitu suureksi. Lisäksi tähän ratkaisuun kohdistuu CRM:n versiopäivityksistä aiheutuva testikirjaston vanhenemisriski.

Ratkaisumallia ehdittiin testaamaan vain hyvin alustavasti. Näissä testeissä havaittiin myös toiminnallisia ongelmia. Ratkaisun toimivuuden varmistamiseksi työssä esitettiin laajempaa konseptitestausta ennen lopullisen testikirjaston toteuttamispäätöksen tekemistä. Vaikka konseptitestaus osoittautuisi toimivaksi, voi testikirjaston toteutus olla riskialtis ja työmäärältään suuri. Työn tulosten kannalta olisi ollut perusteltua tutkia laajemmin kaupallisten testiohjelmistojen tarjoamia mahdollisuuksia saman toiminnallisuuden toteuttamiseksi ja verrata niiden kustannuksia ehdotettuun lähestymistapaan.

7.2 Työn arviointi

Diplomityön alkuperäinen fokus oli keskittyä nimenomaan www-sovelluspalvelukutsujen ja niistä muodostuvien prosessien testaamisen tehostamiseen ylläpitovaiheessa. Regressiotestauksen tehostamiseen ja testiautomaatioon liittyvään teoriaan tutustuessa selvisi kuitenkin, että tämän osa-alueen ratkaiseminen ja testien automatisointi ei olisi ainoana toimenpiteenä järkevää projektin kokonaisuuden kannalta.

Koska projektissa oli tunnistettu jo ennen työn aloittamista muitakin haasteita, jotka vaikuttivat jo käynnissä olevaan kehitystyöhön ja tulisivat jatkumaan myös ylläpitovaiheessa, oli perusteltua laajentaa tutkimusta laaja-alaisemmaksi ja selvittää, mitä muita ylläpitoon ja testaamiseen liittyviä tehtäviä olisi mahdollista tehostaa. Oleellisin tavoite on kuitenkin, että ylläpito ja sen aikana suoritettava testaus on kokonaisuudessaan mahdollisimman tehokasta ja siten mahdollista suorittaa pienemmillä henkilöresursseilla kuin varsinainen toimitusprojekti.

Työn laajeneminen oli työn tekijän osaamisen kehittymisen kannalta erinomainen asia, vaikkakin osa tehdystä kirjallisuustutkimuksesta oli lopputulosten kannalta tarpeetonta. Tämä koskee etenkin regressiotestien valintaan ja priorisointiin liittyviä menetelmiä. Tulevia projekteja ajatellen tästä aihepiiristä saatu tieto vaikuttaa kuitenkin erittäin hyödylliseltä. Toivottavasti työhön kertynyt laaja lähdeluettelo helpottaa myöhemmin muitakin aiheeseen perehtyviä opiskelijoita regressiotestaukseen ja testauksen automatisointiin liittyvässä tiedonhaussa.

Aiheen laajeneminen aiheutti kuitenkin sen, että työn aikana suunniteltujen teknisten ratkaisujen toteuttaminen jäi tulevaisuuteen, eikä niistä voitu esittää työssä yksityiskohtaisempia suunnitelmia tai esimerkkejä. Myöskään tulosten hyödyllisyyttä ei kyetty varmistamaan mittaamalla, mikä on diplomityön laadun

kannalta selvä puute. Toinen puute on suunnitelmien alustavasta laadusta johtuva mittareiden puuttuminen. Tehostamista on lopulta vaikeaa havaita, jos muutosta ei voida mitata. Toisaalta mittareiden laatiminen ei ole täysin ajankohtaista, koska kehitysvaiheessa tehdyt mittaukset eivät välttämättä olisi vertailukelpoisia ylläpitovaiheessa tehtyjen kanssa.

Tehostamisen kannalta työssä otettiin askelia vain testien suorituksen automatisointiin, mutta oleellista olisi myös näiden testien raportoinnin automatisointi.

7.3 Jatkokehityssuunnitelmia

Työssä esitettyjen työkalujen ja muutosten lopullinen toteutus priorisoidaan toteutettavaksi projektin tulevissa kehitysvaiheissa vuoden 2010 aikana. Kaikkia esitettyjä toimenpiteitä ei välttämättä tulla suorittamaan ainakaan työssä esitetyssä laajuudessa. Priorisointia tarkennetaan projektiryhmän ja asiakkaan kanssa kevään 2010 aikana ja toteutuksia tehdään valituilta osin ennen vuoden 2010 lopussa tapahtuvaa hakemus- ja hoitojärjestelmän käyttöönottoa. Alustava priorisointi esitettiin luvussa 6.6.

Käyttöliittymätestauksen automatisointi olisi testien monimutkaisuuden vuoksi varsin hyödyllistä jo testauksessa tapahtuvien virheiden välttämiseksi, mutta taustalla olevan asiakkuudenhallintajärjestelmän aiheuttaman kompleksisuuden vuoksi tämä voi osoittautua liian raskaaksi toimenpiteeksi etenkin tulosten vertailun osalta. Testikirjaston rakentaminen CRM:ää tukevaksi olisi joka tapauksessa järkevää, mutta työmääränsä vuoksi asian toteutus voisi olla järkevämpää toteuttaa keskitetysti koko yrityksen toimesta, jolloin testityökalu saataisiin yrityksen kaikkien projektien käyttöön. Tällöin sen aiheuttamat kustannuksetkin kohdistuisivat tasaisesti perusratkaisun osalta, ja projekti-kohtaisesti jouduttaisiin toteuttamaan vain mahdollisesti tarvittavat laajennukset.

Mikäli toiminnallisten testitapausten luominen ACTSin avulla todetaan toimivaksi menetelmäksi, kannattaa suunniteltua muunnosohjelmaa laajentaa kahdella uudella tavalla myöhemmissä vaiheissa. Ensimmäisenä olisi tuki automaattisen BizUnit-testitapauksen luomiselle. Tällä tarkoitetaan sitä, että testiaineistojen luominen yhdistettäisiin osaksi testiaskelta ilman, että jokainen luotu testitapaus täytyy tallettaa ja suorittaa erikseen. Toinen kohde on peräkkäistiedostojen luomismahdollisuuden lisääminen muuntimeen XML-aineistojen lisäksi. Tässä voitaisiin hyödyntää projektissa aiemmin toteutettuja BizTalk-komponentteja, joilla XML-aineisto saadaan muunnettua peräkkäistiedostoksi myös erittäin monimutkaisten kuvausten kohdalla.

Työstä puuttuivat myös mittarit tehostamistoimenpiteiden arvioimiseksi. Nämä on mietittävä projektijohdon kanssa kuntoon ennen menetelmien käyttöönottoa.

Tieteelliseltä kannalta skeemamuutosten vaikutusten havaitseminen näyttäisi kaipaavan kipeästi lisätutkimusta, koska aihe liittyy läheisesti ulkoisia palvelu-rajapintoja käyttävien järjestelmien ylläpitoon, eikä aiheesta löytynyt kuin muutama konferenssijulkaisu. Näissä tapauksissa vaikutusten analysoinnin automatisointi on keskeinen ongelma, ei muutamissa tutkimuksissa käsitelty XML-dokumenttien automaattinen päivittäminen. Coatesin ja Duin (2010) artikkeli oli erittäin kiinnostavalta vaikuttava lähestymistapa tähän ongelmaan.

8 Yhteenveto

Tässä diplomityössä esiteltiin Merimieseläkekassan järjestelmäuudistusprojektiin liittyvien järjestelmien ylläpitoa tehostavien toimenpiteiden ja työkalujen laatimista. Alkuperäinen www-sovelluspalvelurajapintojen testaamiseen ja testien automatisointiin liittynyt rajaus laajeni työn aikana johtuen ympäristössä havaituista asiaan vaikuttavista seikoista.

Aiheen laajenemisesta johtuen työhön liittyi laaja kirjallisuuskatsaus, jonka tarkoituksena oli lisätä ymmärtämystä testaukseen liittyvästä laajasta ongelmakentästä ja miettiä erilaisten ratkaisumallien ja menetelmien soveltuvuutta tähän projektiin. Osa selvityksestä kohdistui asioihin, joista ei ollut lopulta suurta käytännön hyötyä, mutta tätä ei tiedetty ennakkoon. Näitä olivat etenkin regressiotestien valinta- ja priorisointimenetelmät, joiden kohdalla käyttöönoton esteiksi nousivat kehitysprosessin hallinnassa käytettävien järjestelmien väliset tietokatkokset. Toisaalta tästä aiheesta saatu tieto on erittäin arvokasta toimintatapojen kehittämiseksi tulevissa projekteissa.

Toimintatapojen muutoksia koskevien ehdotusten lisäksi työn aikana löydettiin ratkaisuja tiettyihin projektin erityisongelmiin. Osa näistä menetelmistä ja työkaluista on käytettävissä yleisemminkin, sillä ne perustuvat pääosin avoimen lähdekoodin ratkaisuihin.

BizUnit-testiohjelmisto osoittautui erittäin käyttökelpoiseksi alustaksi myös BizTalk Server -sovelluksen ulkopuolisessa toiminnallisessa testauksessa valmiiden toiminnallisuuksiensa, laajennettavuutensa ja avainsanapohjaisuutensa ansiosta. MEK-projektin kannalta merkittävin ominaisuus oli hyvä tuki www-sovelluspalvelurajapintojen kutsuihin yhdistettynä Windows-alustalle oleelliseen WCF-tukeen. Valmiit komponentit antoivat hyvän lähtökohdan omien laajennusten suunnittelulle. Laajennusten tarkoitus on helpottaa testien toteutusta entisestään ja mahdollistaa testaaminen myös ilman ohjelmointitaitoja.

Palvelurajapintojen jäljittelyyn löytyi vastaava helposti käyttöönotettava ja laajennettava MockingBird-alusta, jolla kehitys- ja testiympäristön herkkyyttä ulkoisten järjestelmien häiriöille voidaan pienentää. Rajapintojen skeemamuutosten tunnistamista helpottamaan kehitettiin prosessi, joka hyödyntää vapaan lähdekoodin XMLZebra-ohjelmaa skeeman sisältämien sanomarakenteiden kuvaamiseen, ilmaista CSDiff-ohjelmaa muutosten havaitsemiseen ja omaa ohjelmaa, joka tutkii eroavien kohtien esiintymistä MEKin liittymien läpi kulkevissa sanomissa. Tällä työkalulla arvioidaan olevan suuri merkitys muutosten vaikutusten analysoinnissa projektin jatkovaiheissa.

Projektin aikataulusta johtuen työn tuloksien toteutus ja tarkempi arviointi sekä mittaus jäivät diplomityön jälkeen tehtäväksi.

Lähteet

Agrawal, H. & Horgan, J.R. & Krauser, E.W. & London, S.A. 1993. *Incremental regression testing*. Teoksessa: *Proceedings of Conference on Software Maintenance 1993. Montreal, Quebec, Canada. 27.-30.9.1993*. USA: IEEE Computer Society. DOI: 10.1109/ICSM.1993.366927.

Ali, A. & Nadeem, A. & Iqbal, M.Z.Z. & Usman, M. 2007. *Regression Testing Based on UML Design Models*. Teoksessa: *Proceedings. 13th Pacific Rim International Symposium on Dependable Computing PRDC 2007. Melbourne, Victoria, Australia. 17-19.12.2007*. USA: IEEE Computer Society. DOI: 10.1109/PRDC.2007.53.

Alspaugh, S. & Walcott, K.R. & Belanich, M. & Kapfhammer, G.M. & Soffa, M.L. 2007. *Efficient time-aware prioritization with knapsack solvers*. Teoksessa: *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007. Atlanta, GA, USA. 5.11.2007*. New York, NY, USA: ACM. DOI: 10.1145/1353673.1353676.

ApTest. 2010. *Software QA Testing and Test Tool Resources*. [Verkkodokumentti]. [Viitattu 21.4.2010]. Saatavilla: http://www.aptest.com/resources.html.

Arcuri, A. & Yao, X. 2007. *On Test Data Generation of Object-Oriented Software*. Teoksessa: Proceedings. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007. Windsor, Iso-Britannia. 12.-14.9.2007. USA: IEEE Computer Society. DOI: 10.1109/TAIC.PART.2007.11.

Arek. 2010. *Arek Oy:n verkkopalvelu*. [Verkkosivusto]. [Viitattu 23.3.2010]. Saatavilla: http://www.arek.fi.

Arisholm, E. & Briand, L.C. & Hove, S.E. & Labiche, Y. 2006. *The impact of UML documentation on software maintenance: an experimental evaluation*. IEEE Transactions on Software Engineering. Vol 32:6. USA: IEEE Computer Society. DOI: 10.1109/TSE.2006.59. ISSN: 0098-5589.

Bach, J. 1999. *Test Automation Snake Oil. V2.1*. [Verkkodokumentti]. [Viitattu 12.3.2010] Saatavilla: http://www.satisfice.com/articles/test_automation_snake_oil.pdf.

Baradhi, G. & Mansour, N. 1997. *A comparative study of five regression testing algorithms*. Teoksessa: *Proceedings of 1997 Australian Software Engineering Conference. Sydney, NSW, Australia. 29.9.-2.10.1997*. USA: IEEE Computer Society. DOI: 10.1109/ASWEC.1997.623769.

Barnett, M. & Fahndrich, M. & de Halleux, P. & Logozzo, F. & Tillmann, N. 2009. *Exploiting the synergy between automated-test-generation and programming-by-contract*. Teoksessa: *Proceedings. ICSE-Companion 2009. IEEE 31st International Conference on Software Engineering - Companion Volume. Vancouver, BC, Kanada. 16.-24.5.2009*. USA: IEEE Computer Society. DOI: 10.1109/ICSE-COMPANION.2009.5071032.

Bartolini, C. & Bertolino, A. & Elbaum, S. & Marchetti, E. 2009a. *Whitening SOA testing*. Teoksessa: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. Amsterdam, Alankomaat. 24.-28.8.2009*. New York, NY, USA: ACM. DOI: 10.1145/1595696.1595721.

Bartolini, C. & Bertolino, A. & Marchetti, E. & Polini, A. 2009b. *WS-TAXI: A WSDL-based Testing Tool for Web Services*. Teoksessa: Proceedings. Second International Conference on Software Testing, Verification, and Validation. ICST 2009. Denver, CO, USA. 1.-4.4.2009. USA: IEEE Computer Society. DOI: 10.1109/ICST.2009.28.

Beck, K. 1999. *Embracing Change with Extreme Programming*. Computer. Vol 32:10. USA: IEEE Computer Society. DOI: 10.1109/2.796139. ISSN 0018-9162.

Beizer, B. 1990. *Software testing techniques, 2nd ed*. New York, NY, USA: Van Nostrand Reinhold. 550 s. ISBN 0-442-20672-0.

Berner, S. & Weber, R. & Keller, R.K. 2005. *Observations and Lessons Learned from Automated Testing*. Teoksessa: Proceedings of the 27th international conference on Software engineering. St. Louis, MO, USA. 15.-21.5.2005. New York, NY, USA: ACM. DOI: 10.1145/1062455.1062556.

Bertolino, A. & Gao, J. & Marchetti, E. & Polini, A. 2007. *TAXI – A Tool for XML-Based Testing*. Teoksessa: *Proceedings. 29th International Conference on Software Engineering. ICSE 2007 Companion Volume. Minneapolis, MN, USA. 20.-26.5.2007*. USA: IEEE Computer Society. DOI: 10.1109/ICSECOMPANION.2007.72.

Bertolino, A. & De Angelis, G. & Frantzen, L. & Polini, A. 2008. *Model-Based Generation of Testbeds for Web Services.* Teoksessa: *Testing of Software and Communicating Systems. Proceedings. 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008. 8th International Workshop, FATES 2008. Tokio, Japani. 10.-13.6.2008.* Heidelberg, Saksa: Springer-Verlag Berlin. DOI: 10.1007/978-3-540-68524-1_19.

Bertolino, A. 2009. *Approaches to testing service-oriented software systems.* Teoksessa: *Proceedings of the 1st international workshop on Quality of service-oriented software systems. Amsterdam, Alankomaat. 25.8.2009.* New York, NY, USA: ACM. DOI: 10.1145/1596473.1596475.

Bieberstein, N & Bose, S. & Fiammante, M. & Jones, K. & Shah, R. 2006. *Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap.* Upper Saddle River, NJ, USA: Pearson Education Inc. 272 s. ISBN 0-13-187002-5. Saatavilla myös sähköisesti: http://books24x7.com/toc.asp?bookid=12241.

Blackburn, M. & Busser, R. & Nauman, A. 2004. *Why model-based test automation is different and what you should know to get started.* International Conference on Practical Software Quality and Testing, 2004. [Verkkodokumentti]. [Viitattu 15.3.2010]. Saatavilla: http://www.psqtconference.com/2004east/tracks/Tuesday/PSTT_2004_blackburn.pdf.

Boehm, B. 1984. *Verifying and Validating Software Requirements and Design Specifications.* IEEE Software. Vol 1:1. USA: IEEE Computer Society. DOI: 10.1109/MS.1984.233702. ISSN 0740-7459.

Boehm, B. & Basili, V.R. 2001. *Software Defect Reduction Top 10 List.* Computer. Vol 34:1. USA: IEEE Computer Society. DOI: 10.1109/2.962984. ISSN 0018-9162.

Boehm, B. 2002. *Get ready for agile methods, with care.* Computer. Vol 35:1. USA: IEEE Computer Society. DOI: 10.1109/2.976920. ISSN 0018-9162.

Briand, L.C. & Labiche, Y. & He, S. 2009. *Automating regression test selection based on UML designs.* Information and Software Technology. Vol 51:1. Amsterdam, Hollanti: Elsevier B.V. DOI: 10.1016/j.infsof.2008.09.010. ISSN 0950-5849.

Bruno, M. & Canfora, G. & Di Penta, M. & Esposito, G. & Mazza, V. 2005. *Using Test Cases as Contract to Ensure Service Compliance Across Releases.* Teoksessa: *Proceedings. Service-Oriented Computing - ICSOC 2005. Amsterdam, Alankomaat. 12.-15.12.2005.* Heidelberg, Saksa: Springer-Verlag Berlin. DOI: 10.1007/11596141_8.

Burnstein, I. 2003. *Practical software testing: a process-oriented approach.* New York, NY, USA: Springer-Verlag New York, Inc. 709 s. ISBN 0-387-95131-8 (painettu). ISBN 0-387-21658-8 (sähköinen).

Canfora, G. & Di Penta, M. 2006. *Testing services and service-centric systems: challenges and opportunities.* IT Professional. Vol 8:2. USA: IEEE Computer Society. DOI: 10.1109/MITP.2006.51. ISSN 1520-9202.

Chang, T.-H. & Yeh, T. & Miller, R.C. 2010. *GUI Testing Using Computer Vision.* Julkaistaan teoksessa: *Proceedings of the 28th international conference on Human factors in computing systems. CHI 2010. Atlanta, GA, USA. 10.-15.4.2010.* New York, NY, USA: ACM. [Viitattu 3.4.2010]. Saatavilla: http://groups.csail.mit.edu/uid/projects/sikuli/sikuli-chi2010.pdf.

Chen, Y. & Rosenblum, D.S. & Vo, K-P. 1994. *TESTTUBE: a system for selective regression testing.* Teoksessa: *Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italia. 16.-21.5.1994.* USA: IEEE Computer Society / ACM. DOI: 10.1109/ICSE.1994.296780.

Chen, Y. & Probert, R.L. & Sims, D.P. 2002. *Specification-based regression test selection with risk analysis.* Teoksessa: *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research. Toronto, Ontario, Canada. 30.9.-3.10.2002.* USA: IBM Press. [Viitattu 31.1.2010]. Saatavilla: http://www.site.uottawa.ca/~ychen/CASCON02.pdf.

Coates, A.B. & Dui, D. 2010. *"Full Impact" Schema Diferencing.* Teoksessa: *XML Prague 2010 Conference Proceedings. Praha, Tsekin tasavalta. 13.-14.3.2010.* S. 65-85. Tsekin tasavalta: MATFYZPRESS. [Viitattu 31.3.2010]. ISBN 978-80-7378-115-6. Saatavilla: http://www.xmlprague.cz/2010/files/XMLPrague_2010_Proceedings.pdf.

DeMarco, T & Boehm, B. 2002. *The agile methods fray.* Computer. Vol 35:6. USA: IEEE Computer Society. DOI: 10.1109/MC.2002.1009175. ISSN 0018-9162.

Datz, T. 2004. What You Need to Know About Service-Oriented Architecture. CIO. Vol 17:7. USA: CXO Media Inc. ISSN 0894-9301. Saatavilla myös:

http://www.cio.com/article/32060/What_You_Need_to_Know_About_Service_Oriented_Architecture.

Dustin, E. & Rashka, J. & Paul, J. 1999. *Automated Software Testing: Introduction, Management, and Performance*. Reading, MA, USA: Addison-Wesley. 575 s. ISBN 0-201-43287-0.

Dustin, E. 2002. *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Boston, MA, USA: Addison-Wesley Professional. 271 s. ISBN 0-201-79429-2.

Dustin, E. & Garrett, T. & Gauf, B. 2009. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Boston, MA, USA: Addison-Wesley Professional. 340 s. ISBN 0-321-58051-6.

Elbaum, S. & Malishevsky, A.G. & Rothermel, G. 2002. *Test case prioritization: a family of empirical studies*. IEEE Transactions on Software Engineering. Vol 28:2. USA: CXO Media Inc. DOI: 10.1109/32.988497. ISSN 0098-5589.

Elbaum, S. & Kallakuri, P. & Malishevsky, A. & Rothermel, G. & Kanduri, S. 2003. *Understanding the effects of changes on the cost-effectiveness of regression testing techniques*. Software Testing, Verification and Reliability. Vol 13:2. Malden, MA, USA: John Wiley & Sons Inc. DOI: 10.1002/stvr.263. ISSN 0960-0833.

Engström E. & Runeson, P. & Skoglund, M. 2009. *A systematic review on regression test selection techniques*. Information and Software Technology. Vol 52:1. Amsterdam, Hollanti: Elsevier B.V. DOI: 10.1016/j.infsof.2009.07.001. ISSN 0950-5849.

Erl, T. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR. 792 s. ISBN 0-13-185858-0.

ETK. 2010a. Työeläke.fi-verkkopalvelu. [Verkkosivusto]. [Viitattu 23.3.2010]. Saatavilla: http://www.tyoelake.fi.

ETK. 2010b. Eläketurvakeskuksen verkkopalvelu. [Verkkosivusto]. [Viitattu 23.3.2010]. Saatavilla: http://www.etk.fi.

Fagan, M.E. 1999. *Design and code inspections to reduce errors in program development*. IBM Systems Journal. Vol 38:2/3. S. 258-287. USA: IBM. ISSN 0018-8670.

Fewster, M. 2001. *Common Mistakes in Test Automation*. Fall Test Automation Conference 2001. [Verkkodokumentti]. [Viitattu 12.3.2010] Saatavilla: http://www.scionlabs.com/pdf-commonmistakesintaut.pdf.

Fewster M. & Graham D. 1999. *Software Test Automation: Effective use of test execution tools*. Harlow, Iso-Britannia: Pearson Education Limited. 574 s. ISBN 978-0-201-33140-0.

Gittens, M. & Lutfiyya, H. & Bauer, M. & Godwin, D. & Kim, Y.W. & Gupta, P. 2002. *An empirical evaluation of system and regression testing*. Teoksessa: *Proceedings of the 2002 conference of the Center for Advanced Studies on Collaborative research. Toronto, Ontario, Canada. 30.9.-3.10.2002*. USA: IBM Press. Saatavilla: http://portal.acm.org/ft_gateway.cfm?id=782118

Gittens, M. & Romanufa, K. & Godwin, D. & Racicot, J. 2006. *All code coverage is not created equal: a case study in prioritized code coverage*. Teoksessa: *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research. Toronto, Ontario, Canada. 16.-19.10.2006*. New York, NY, USA: ACM. DOI: 10.1145/1188966.1188981.

Gorthi, R.P. & Pasala, A. & Chanduka, K.K.P. & Leong, B. 2008. *Specification-Based Approach to Select Regression Test Suite to Validate Changed Software*. Teoksessa: *Proceedings. 15th Asia-Pacific Software Engineering Conference APSEC 2008. Beijing, Kiina. 2.-5.12.2008*. USA: IEEE Computer Society. DOI: 10.1109/APSEC.2008.56.

Graham, D. & van Veenendal, E. & Evans, I. & Black, R. 2007. *Foundations of Software Testing: ISTQB Certification*. [Sähköinen kirja]. USA: Cengage Learning & Books24x7. Saatavilla: http://common.books24x7.com/book/id_26179/book.asp. ISBN 978-1-84480-355-2 (painettu).

Graves, T.L. & Harrold, M.J. & Kim, J-M. & Porter, A. & Rothermel, G. 2001. *An empirical study of regression test selection techniques*. ACM Transactions on Software Engineering and Methodology. Vol 10:2. New York, NY, USA: ACM. DOI: 10.1145/367008.367020. ISSN 1049-331X.

Guerrini, G. & Mesiti, M. & Rossi, D. 2005. *Impact of XML schema evolution on valid documents*. Teoksessa: *Proceedings of the 7th annual ACM international workshop on Web information and data management. Bremen, Saksa. 4.11.2005*. New York, NY, USA: ACM. DOI: 10.1145/1097047.1097056.

Gupta, R. & Harrold, M.J. & Soffa, M.L. 1992. *An approach to regression testing using slicing.* Teoksessa: *Proceedings of Conference on Software Maintenance 1992. Orlando, FL, USA. 9.-12.11.1992.* USA: IEEE Computer Society. DOI: 10.1109/ICSM.1992.242531.

Hamlet, D. 2006. *Subdomain testing of units and systems with state.* Teoksessa: *Proceedings of the 2006 international symposium on Software testing and analysis. Portland, ME, USA. 17.-20.7.2006.* New York, NY, USA: ACM. DOI: 10.1145/1146238.1146249.

Harjumaa, L. & Tervonen, I. & Huttunen, A. 2006. *Peer reviews in real life - motivators and demotivators.* Teoksessa: *Proceedings. Fifth International Conference on Quality Software. QSIC 2009. Melbourne, Australia. 19.-20.9.2005.* USA: IEEE Computer Society. DOI: 10.1109/QSIC.2005.48.

Harrold, M.J. & Soffa, M.L. 1988. *An Incremental Approach to Unit Testing during Maintenance.* Teoksessa: *Proceedings of the Conference on Software Maintenance 1988. Phoenix, AZ, USA. 24.-27.10.1988.* USA: IEEE Computer Society. DOI: 10.1109/ICSM.1988.10188.

Harrold, M.J. & Jones, J.A. & Li, T. & Liang, D. & Orso, A. & Pennings, M. & Sinha, S. & Spoon, S.A. & Gujarathi. 2001. *A Regression test selection for Java software.* Teoksessa: *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications 2001. Tampa Bay, FL, USA. 14.-18.10.2001.* New York, NY, USA: ACM. DOI: 10.1145/504282.504305.

Harrold, M.J. & Orso, A. 2008. *Retesting software during development and maintenance.* Teoksessa: *Proceedings of the 2008 Frontiers of Software Maintenance. Beijing, Kiina. 28.9.-4.10.2008.* USA: IEEE Computer Society. DOI: 10.1109/FOSM.2008.4659253.

Harrold, M.J. 2009. *Reduce, reuse, recycle, recover: Techniques for improved regression testing.* Teoksessa: *Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM). Edmonton, Alberta, Canada. 20.-26.9.2009.* USA: IEEE Computer Society. DOI: 10.1109/ICSM.2009.5306347.

Hartmann, J. & Robson, D.J. 1990. *Techniques for selective revalidation.* IEEE Software. Vol 7:1. USA: IEEE Computer Society. DOI: 10.1109/52.43047. ISSN 0740-7459.

Hartmann, J. & Imoberdorf, C. & Meisinger, M. 2000. *UML-Based integration testing.* Teoksessa: *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis. Portland, OR, USA. 21.-24.8.2000.* New York, NY, USA: ACM. DOI: 10.1145/347324.348872.

Hsu, H-Y. & Orso, A. 2009. *MINTS: A general framework and tool for supporting test-suite minimization.* Teoksessa: *Proceedings. ICSE 2009. IEEE 31st International Conference on Software Engineering. Vancouver, BC, Kanada. 16.-24.5.2009.* USA: IEEE Computer Society. DOI: 10.1109/ICSE.2009.5070541.

Huo, M. & Verner, J. & Liming Zhu & Babar, M.A. 2004. *Software Quality and Agile Methods.* Teoksessa: *Proceedings of the 28th Annual International Computer Software and Applications Conference: COMPSAC 2004. Hong Kong. 28.-30.9.2004.* USA: IEEE Computer Society. DOI: 10.1109/CMPSAC.2004.1342889.

IEEE 610.12-1990. 1990. *IEEE standard glossary of software engineering terminology.* USA: Standards Coordinating Committee of the Computer Society of the IEEE. DOI: 10.1109/IEEESTD.1990.101064.

IEEE 829-2008. 2008. *IEEE Standard for Software and System Test Documentation.* USA, Software & Systems Engineering Standards Committee of the IEEE Computer Society. DOI: 10.1109/IEEESTD.2008.4578383.

IEEE 1008-1987. 1986. *IEEE Standard for Software Unit Testing.* USA: Software Engineering Standards Subcommittee of the Software Engineering Technical Committee of the IEEE Computer Society. DOI: 10.1109/IEEESTD.1986.81001.

IEEE 1059-1993. 1993. *IEEE Guide for Software Verification and Validation Plans.* USA: Software Engineering Standards Committee of the IEEE Computer Society. DOI: 10.1109/IEEESTD.1994.121430.

Jones, J.A. & Harrold, M.J. 2003. *Test-suite reduction and prioritization for modified condition/decision coverage.* IEEE Transactions on Software Engineering. Vol 29:3. USA: IEEE Computer Society. DOI: 10.1109/TSE.2003.1183927. ISSN 0098-5589.

Kaner, C. & Falk, J. & Nguyen, H.Q. 1999. *Testing computer software. 2nd ed.* New York, NY, USA: John Wiley & Sons, Inc. 480 s. ISBN 0-471-35846-0.

Kaner, C. & Bach, J. & Pettichord, B. 2002. *Lessons Learned in Software Testing: A Context-Driven Approach*. New York, NY, USA: John Wiley & Sons, Inc. 286 s. ISBN 0-471-08112-4.

Karhu, K. & Repo, T. & Taipale, O. & Smolander, K. 2009. *Empirical Observations on Software Testing Automation*. Teoksessa: *Proceedings. International Conference on Software Testing Verification and Validation. ICST '09. Denver, CO, USA. 1.-4.4.2009*. USA: IEEE Computer Society. DOI: 10.1109/ICST.2009.16.

Kim, Y.W. 2003. *Efficient use of code coverage in large-scale software development*. Teoksessa: *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research. Toronto, Ontario, Canada. 6.-9.10.2003*. USA: IBM Press. Saatavilla: http://portal.acm.org/ft_gateway.cfm?id=961347.

King, J.C. 1975. *A new approach to program testing*. ACM SIGPLAN Notices. Vol 10:6. New York, NY, USA: ACM. DOI: 10.1145/390016.808444. ISSN 0362-1340.

Koju, T. & Takada, S. & Doi, N. 2003. *Regression test selection based on intermediate code for virtual machines*. Teoksessa: *Proceedings International Conference on Software Maintenance ICSM 2003. Amsterdam, Alankomaat. 22.-26.9.2003*. USA: IEEE Computer Society. DOI: 10.1109/ICSM.2003.1235452.

Korel, B. & Al-Yami, A.L. 1998. *Automated regression test generation*. Teoksessa: *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis. Clearwater Beach, FL, USA. 2.-4.3.1998*. New York, NY, USA: ACM. DOI: 10.1145/271771.271803.

Korel, B. & Koutsogiannakis, G. 2009. *Experimental Comparison of Code-Based and Model-Based Test Prioritization*. Teoksessa: *IEEE International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. Denver, CO, USA. 1.-4.4.2009*. USA: IEEE Computer Society. DOI: 10.1109/ICSTW.2009.45.

Kuhn, R. & Kacker, R. & Yu Lei & Hunter, J. 2009. *Combinatorial Software Testing*. Computer. Vol 42:8. USA: IEEE Computer Society. DOI: 10.1109/MC.2009.253. ISSN 0018-9162.

Kung, D. & Gao, J. & Hsia, P & Lin, J. & Toyoshima, Y. 1993. *Class firewall, test order and regression testing of object-oriented programs*. Journal of Object-Oriented Programming. Vol 8:2. S. 51-65. [Viitattu 23.1.2010]. ISSN 0896-8438. Saatavilla: http://citeseer.ist.psu.edu/cache/papers/cs/2181/ftp:zSzzSzpepe.uta.eduzSzpubzSzpublicationszSzjss.pdf/kung93class.pdf.

Labiche, Y. & Thévenod-Fosse, P. & Waeselynck, H. & Durand, M.-H. 2000. *Testing levels for object-oriented software*. Teoksessa: *Proceedings of the 2000 International Conference on Software Engineering. Limerick, Ireland. 4.-11.6.2000*. New York, NY, USA: ACM. DOI: 10.1109/ICSE.2000.870405.

Leonardi, E. & Hoai, T.T. & Bhowmick, S.S & Madria, S. 2007. *DTD-Diff: A change detection algorithm for DTDs*. Data & Knowledge Engineering. Vol 61:2. Amsterdam, Hollanti: Elsevier B.V. DOI: 10.1016/j.datak.2006.06.003. ISSN 0169-023X.

Leung, H.K.N. & White, L. 1989. *Insights into Regression Testing*. Teoksessa: *Proceedings of Conference on Software Maintenance 1989. Miami, FL, USA. 16.-19.10.1989*. USA: IEEE Computer Society. DOI: 10.1109/ICSM.1989.65194.

Leung, H.K.N. & White, L. 1990. *A study of integration testing and software regression at the integration level*. Teoksessa: *Proceedings of Conference on Software Maintenance 1990. San Diego, CA, USA. 26.-29.11.1990*. USA: IEEE Computer Society. DOI: 10.1109/ICSM.1990.131377.

Leung, H.K.N. & White, L. 1991. *A Cost Model to Compare Regression Test Strategies*. Teoksessa: *Proceedings of Conference on Software Maintenance 1991. Sorrento, Italy. 15.-17.10.1991*. USA: IEEE Computer Society. DOI: 10.1109/ICSM.1991.160330.

Li, Z. & Harman, M. & Hierons, R.M. 2007. *Search Algorithms for Regression Test Case Prioritization*. IEEE Transactions on Software Engineering. Vol 33:4. USA: IEEE Computer Society. DOI: 10.1109/TSE.2007.38. ISSN 0098-5589.

Mansour, N. & El-Fakih, K. 1997. *Natural optimization algorithms for optimal regression testing*. Teoksessa: *Proceedings of The Twenty-First Annual International Computer Software & Applications Conference (COMPSAC'97). Washington, DC, USA. 13.-15.8.1997*. USA: IEEE Computer Society. DOI: 10.1109/CMPSAC.1997.625060.

Mansour, N. & Bahsoon, R. & Baradhi, G. 2001. *Empirical comparison of regression test selection algorithms*. Journal of Systems and Software. Vol 57:1. Amsterdam, Hollanti: Elsevier B.V. DOI: 10.1016/S0164-1212(00)00119-9. ISSN 0164-1212.

Mao, C. & Lu, Y. 2005. *Regression testing for component-based software systems by enhancing change*. Teoksessa: *Proceedings. 12th Asia-Pacific Software Engineering Conference. Taipei, Taiwan. 15.-17.12.2005*. USA: IEEE Computer Society. DOI: 10.1109/APSEC.2005.95.

Martin, R.C. 2005. *The test bus imperative: architectures that support automated acceptance testing*. IEEE Software. Vol 22:4. USA: IEEE Computer Society. DOI: 10.1109/MS.2005.110. ISSN 0740-7459.

McCabe, T.J. 1976. *A Complexity Measure*. IEEE Transactions on Software Engineering. Vol SE-2:4. USA: IEEE Computer Society. DOI: 10.1109/TSE.1976.233837. ISSN 0098-5589.

McMinn, P. 2004. *Search-based software test data generation: a survey*. Software Testing, Verification and Reliability. Vol 14:2. Malden, MA, USA: John Wiley & Sons Inc. DOI: 10.1002/stvr.294. ISSN 0960-0833.

Mei, H. & Zhang, L. 2005. *A framework for testing Web services and its supporting tool.* Teoksessa: *Proceedings. IEEE International Workshop on Service-Oriented System Engineering. SOSE 2005. Beijing, Kiina. 20.-21.10.2005*. USA: IEEE Computer Society. DOI: 10.1109/SOSE.2005.1.

MEK. 2009. *Merimieseläkekassa: vuosikertomus 2008*. [Verkkodokumentti]. [Viitattu 23.3.2010]. Saatavilla: http://www.merimieselakekassa.fi/fi/Etusivu/Yleista/Julkaisut/Julkaisut/Vuosikertomus_2008.pdf.

MEK. 2010. *Merimieseläkekassan verkkosivut*. [Verkkosivusto]. [Viitattu 23.3.2010]. Saatavilla: http://www.merimieselakekassa.fi.

MEKAUD. 2010. *MEK tavoitearkkitehtuuri -projektin auditointiraportti*. [Ei-julkinen projektidokumentti].

MEKPS. 2010. *MEK vaihe 4 projektisuunnitelma*. [Ei-julkinen projektidokumentti].

MEKTS. 2010. *MEK vaihe 4 testaussuunnitelma*. [Ei-julkinen projektidokumentti].

Meszaros, G. 2003. *Agile regression testing using record & playback*. Teoksessa: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Anaheim, CA, USA. 26.-30.10.2003*. New York, NY, USA: ACM. DOI: 10.1145/949344.949442.

MS. 2010. *Microsoft Releases Technical Preview of Next Generation of Microsoft Dynamics CRM*. Lehdistötiedote. [Verkkodokumentti]. [Viitattu 21.4.2010]. Saatavilla: http://www.microsoft.com/Presspass/press/2010/mar10/03-25CTP3PR.mspx.

MSDN. 2010a. *Visual Studio Team System: About JavaScript and ActiveX Controls in Web Tests*. [Verkkodokumentti]. [Viitattu 1.4.2010]. Saatavilla: http://msdn.microsoft.com/en-us/library/ms404678.aspx.

MSDN. 2010b. *Common Scenarios for Creating Coded Web Tests with Team System 2008*. [Verkkodokumentti]. [Viitattu 2.4.2010]. Saatavilla: http://msdn.microsoft.com/en-us/library/cc681342.aspx

Myers, G. 1979. *The Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc. 177 s. ISBN 0-471-04328-1.

Myers, G. & Badgett, T. & Thomas, T. & Sandler, C. 2004. *The Art of Software Testing. 2nd ed. Revised and Updated by Tom Badgett and Todd M. Thomas with Corey Sandler*. Hoboken, NJ, USA: John Wiley & Sons, Inc. 234 s. ISBN 0-471-46912-2.

Nagle, C.J. 2002. *Test Automation Frameworks*. [Verkkodokumentti]. [Viitattu 13.3.2010] Saatavilla: http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm

Nebut, C. & Fleurey, F. & Le Traon, Y. & Jezequel, J.-M. 2006. *Automatic test generation: a use case driven approach*. IEEE Transactions on Software Engineering. Vol 32:3. USA: IEEE Computer Society. DOI: 10.1109/TSE.2006.22. ISSN 0098-5589.

OASIS. 2006. *Web Services Security: SOAP Message Security 1.1*. [Verkkodokumentti]. [Viitattu 11.4.2010] Saatavilla: http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

OASIS. 2007. *Web Services Business Process Execution Language Version 2.0*. [Verkkodokumentti]. [Viitattu 11.4.2010] Saatavilla: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

Offutt, J. & Liu, S. & Abdurazik, A. & Ammann, P. 2003. *Generating test data from state-based specifications*. Software Testing, Verification and Reliability. Vol 13:1. Malden, MA, USA: John Wiley & Sons Inc. DOI: 10.1002/stvr.264. ISSN 0960-0833.

Offutt, J. & Xu, W. 2004. *Generating test cases for web services using data perturbation*. ACM SIGSOFT Software Engineering Notes. Vol 29:5. New York, NY, USA: ACM. DOI: 10.1145/1022494.1022529. ISSN 0163-5948.

Opdyke, W. 1992. *Refactoring Object-Oriented Frameworks*. [Verkkodokumentti] Väitöskirja. Graduate College of the University of Illinois at Urbana-Champaign, Computer Science. [Viitattu: 2.1.2010]. Saatavilla: http://www.laputan.org/pub/papers/opdyke-thesis.pdf.

Orso, A. & Harrold, M.J. & Rosenblum, D. & Rothermel, G. & Soffa, M.L. & Do, H. 2001. *Using Component Metacontent to Support the Regression Testing of Component-Based Software*. Teoksessa: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001. Firenze, Italia. 7.-9.11.2001*. USA: IEEE Computer Society. DOI: 10.1109/ICSM.2001.972790.

Orso, A. & Shi, N. & Harrold, M.J. 2004. *Scaling regression testing to large software systems*. Teoksessa: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Newport Beach, CA, USA. 31.10.-5.11.2004*. New York, NY, USA: ACM. DOI: 10.1145/1029894.1029928.

Patton, R. 2000. *Software testing*. Indianapolis, IN, USA: Sams Publishing. 389 s. ISBN 0-672-31983-7.

Pasala, A. & Lew Yaw Fung, Y.L.H, & Akladios, F. & Gorthi, R.P. 2008. *An approach to select regression tests to validate .NET applications upon deployment of components upgrades*. Teoksessa: *Proceedings of the 1st Bangalore annual Compute conference. Bangalore, India. 18.-20.1.2008*. New York, NY, USA: ACM. DOI: 10.1145/1341771.1341778.

Passi, K. & Morgan, D. & Madria, S. 2009. *Maintaining integrated XML schema*. Teoksessa: *Proceedings of the 2009 International Database Engineering & Applications Symposium. Cetraro, Calabria, Italia. 16.-18.9.2009*. New York, NY, USA: ACM. DOI: 10.1145/1620432.1620461.

Piwowarski, P. & Ohba, M. & Caruso, J. 1993. *Coverage measurement experience during function test*. Teoksessa: *ICSE '93: Proceedings of the 15th international conference on Software Engineering. Baltimore, MD, USA. 17.-21.5.1993*. USA: IEEE Computer Society. Saatavilla: http://portal.acm.org/ft_gateway.cfm?id=257635.

Poimala, S. & Heikniemi, J. & Blåfeld, H. 2008. *Ketterät Käytännöt. Iteraatiot ja inkrementit*. [Verkkodokumentti]. [Viitattu 21.3.2010]. Saatavissa: http://www.ketteratkaytannot.fi/.

Rajan, A. 2006. *Coverage Metrics to Measure Adequacy of Black-Box Test Suites*. Teoksessa: *Proceedings. 21st IEEE International Conference on Automated Software Engineering. ASE 2006. Tokio, Japani. 18.-22.9. 2006*. USA: IEEE Computer Society. DOI: 10.1109/ASE.2006.31.

Ramler, R. & Wolfmaier, K. 2006. *Economic perspectives in test automation: balancing automated and manual testing with opportunity cost*. Teoksessa: *Proceedings of the 2006 international workshop on Automation of software test. Shanghai, Kiina. 23.5.2006*. New York, NY, USA: ACM. DOI: 10.1145/1138929.1138946.

Rosenblum, D.S. & Weyuker, E.J. 1997. *Using coverage information to predict the cost-effectiveness of regression testing strategies*. IEEE Transactions on Software Engineering. Vol 23:3. USA: IEEE Computer Society. DOI: 10.1109/32.585502. ISSN 0098-5589.

Rothermel, G. & Harrold, M.J. 1993. A Safe, Efficient Algorithm for Regression Test Selection. Teoksessa: *Proceedings of Conference on Software Maintenance 1993. Montreal, Quebec, Canada. 27.-30.9.1993*. USA: IEEE Computer Society. DOI: 10.1109/ICSM.1993.366926.

Rothermel, G. & Harrold, M.J. 1996. *Analyzing regression test selection techniques*. IEEE Transactions on Software Engineering. Vol 22:8. USA: IEEE Computer Society. DOI: 10.1109/32.536955. ISSN 0098-5589.

Rothermel, G. & Harrold, M.J. 1997. *A safe, efficient regression test selection technique*. IEEE Transactions on Software Engineering and Methodology. Vol 6:2. USA: IEEE Computer Society. DOI: 10.1145/248233.248262. ISSN 1049-331X.

Rothermel, G. & Harrold, M.J. & Dedhia, J. 2000. *Regression Test Selection for C++ Software*. Journal of Software Testing, Verification, and Reliability. Vol 10:2. Malden, MA, USA: John Wiley & Sons Inc. DOI: 10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E. ISSN 0960-0833.

Rothermel, G. & Untch, R.H. & Harrold, M.J. 2001. *Prioritizing Test Cases For Regression Testing*. IEEE Transactions on Software Engineering. Vol 27:10. USA: IEEE Computer Society. DOI: 10.1109/32.962562. ISSN 0098-5589.

Sajeev, A.S.M. & Wibowo, B. 2003. *Regression test selection based on version changes of components*. Teoksessa: *Tenth Asia-Pacific Software Engineering Conference. APSEC 2003. Chiang Mai, Thaimaa. 10.-12.12.2003*. USA: IEEE Computer Society. DOI: 10.1109/APSEC.2003.1254360.

Samuel, P. & Mall, R. & Bothra, A.K. 2008. *Automatic test case generation using unified modeling language (UML) state diagrams*. IET Software. Vol 2:2. Stevenage, Iso-Britannia: Institution of Engineering and Technology. DOI: 10.1049/iet-sen:20060061. ISSN 1751-8806.

Sapna, P.G. & Mohanty, H. 2009a. *Prioritization of Scenarios Based on UML Activity Diagrams*. Teoksessa: *First International Conference on Computational Intelligence, Communication Systems and Networks, 2009. CICSYN '09. Indore, Intia. 23.-25.7.2009*. USA: IEEE Computer Society. DOI: 10.1109/CICSYN.2009.74.

Sapna, P.G. & Mohanty, H. 2009b. *Prioritizing Use Cases to aid ordering of Scenarios*. Teoksessa: *Proceedings 2009 Third UKSim European Symposium on Computer Modeling and Simulation. EMS '09. Ateena, Kreikka. 25.-27.11.2009*. USA: IEEE Computer Society. DOI: 10.1109/EMS.2009.78.

Skoglund, M. & Runeson, P. 2005. *A case study of the class firewall regression test selection technique on a large scale distributed software system*. Teoksessa: *Proceedings of 2005 International Symposium on Empirical Software Engineering. Noosa Heads, Queensland, Australia. 17.-18.11.2005*. USA: IEEE Computer Society. DOI: 10.1109/ISESE.2005.1541816.

Sommerville, I. 2004. *Software engineering. 7th ed*. Harlow, Iso-Britannia: Pearson Education Limited. 759 s. ISBN 0-321-21026-3.

Sommerville, I. 2007. *Software engineering. 8th ed*. Harlow, Iso-Britannia: Pearson Education Limited. 840 s. ISBN 978-0-321-31379-9.

Sprott, D. & Wilkes, L. 2004. *Understanding Service-Oriented Architecture*. [Verkkodokumentti]. [Viitattu 10.4.2010]. Saatavilla: http://msdn.microsoft.com/en-us/library/aa480021.aspx.

Srikanth, H. & Williams, L. & Osborne, J. 2005. *System test case prioritization of new and regression test cases*. Teoksessa: *Proceedings. 2005 International Symposium on Empirical Software Engineering. Noosa Heads, Queensland, Australia. 17.-18.11.2005*. USA: IEEE Computer Society. DOI: 10.1109/ISESE.2005.1541815.

Srivastava, A. & Thiagarajan, J. 2002. *Effectively prioritizing tests in development environment*. Teoksessa: *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis. Rooma, Italia, 22.-24.7.2002*. New York, NY, USA: ACM. DOI: 10.1145/566172.566187.

Sun, C. & Zhang, B. & Li, J. 2009. *TSGen: A UML Activity Diagram-Based Test Scenario Generation Tool*. Teoksessa: *Proceedings. International Conference on Computational Science and Engineering. CSE '09. Vancouver, Kanada. 29.-31. 8.2009*. USA: IEEE Computer Society. DOI: 10.1109/CSE.2009.99.

Sun, Y. & Jones, E.L. 2004. *Specification-driven automated testing of GUI-based Java programs*. Teoksessa: *Proceedings of the 42nd annual Southeast regional conference. Huntville, AL, USA. 2.-3.4.2004*. New York, NY, USA: ACM. DOI: 10.1145/986537.986570.

SWEBOK. (Bourque, P. & Dupuis, R). 2004. *Guide to the Software Engineering Body of Knowledge 2004 Version*. USA: IEEE Computer Society. 191 s. ISBN 0-7695-2330-7. Saatavilla myös sähköisesti: http://www.computer.org/portal/web/swebok/htmlformat.

Takala, T. & Maunumaa, M. & Katara, M. 2009. *An Adapter Framework for Keyword-Driven Testing*. Teoksessa: *Proceedings. 2009 Ninth International Conference on Quality Software. QSIC 2009. Jeju, Etelä-Korea. 24.-25.8.2009*. USA: IEEE Computer Society. DOI: 10.1109/QSIC.2009.35.

Tang, J. & Cao, X. & Ma, A. 2008. *Towards Adaptive Framework of Keyword Driven Automation Testing*. Teoksessa: *Proceedings. IEEE International Conference on Automation and Logistics. ICAL 2008. Qingdao, Kiina. 1.-3.9.2008*. USA: IEEE Computer Society. DOI: 10.1109/ICAL.2008.4636415.

TTL (Tietotekniikan liitto ry). 2007. *ISTQB:n testaussanasto*. [Verkkodokumentti]. Suomi. [Viitattu 13.12.2009]. Saatavissa: http://www.ttlry.fi/@Bin/14155799/istqb_sanasto.pdf.

Tsai W.T. & Bai, X. & Paul, R. & Yu, L. 2001. *Scenario-Based Functional Regression Testing.* Teoksessa: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001. Chicago, IL, USA. 8.-12.10.2001.* USA: IEEE Computer Society. DOI: 10.1109/CMPSAC.2001.960659.

Tsai, W.T & Paul, R. & Song, W & Cao, Z. 2002. *Coyote: an XML-based framework for Web services testing.* Teoksessa: *Proceedings. 7th IEEE International Symposium on High Assurance Systems Engineering. HASE 2002. Tokio, Japani. 23.-25.10.2002.* USA: IEEE Computer Society. DOI: 10.1109/HASE.2002.1173120.

Volokos, F.I. & Frankl. P.G. 1997. *Pythia: a regression test selection tool based on textual differencing.* Teoksessa: *IFIP TC5 WG5.4 3rd international conference on Reliability, quality and safety of software-intensive systems.* Lontoo, Iso-Britannia: Chapman & Hall, Ltd. [Viitattu 4.3.2010]. Saatavilla: http://cricket.cs.drexel.edu/~filip/Pythia.pdf

Wahl, N.J. 1999. *An Overview of Regression Testing.* ACM SIGSOFT Software Engineering Notes. Vol 24:1. New York, NY, USA: ACM. DOI: 10.1145/308769.308790. ISSN 0163-5948.

Weyuker, E.J. 1998. *Testing component-based software: a cautionary tale.* IEEE Software. Vol 15:5. USA: IEEE Computer Society. DOI: 10.1109/52.714817. ISSN 0740-7459.

White, L. & Leung, H.K.N. 1992. *A Firewall Concept for both Control-Flow and Data-Flow in Regression Integration Testing.* Teoksessa: *Proceedings of Conference on Software Maintenance 1992. Orlando, FL, USA. 9.-12.11.1992.* USA: IEEE Computer Society. DOI: 10.1109/ICSM.1992.242535.

White, L. & Jaber, K. & Robinson, B. 2005. *Utilization of extended firewall for object-oriented regression testing.* Teoksessa: *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05. Budapest, Hungary. 26.-29.9.2005.* USA: IEEE Computer Society. DOI: 10.1109/ICSM.2005.101.

White, L. & Jaber, K. & Robinson, B. & Rajlich, V. 2008. *Extended firewall for regression testing: an experience report.* Journal of Software Maintenance and Evolution: Research and Practice. Vol 20:6. Malden, MA, USA: John Wiley & Sons Inc. DOI: 10.1002/smr.371. ISSN 1532-060X.

Wikipedia. 2009. *List of tools for static code analysis.* [Verkkodokumentti]. [Viitattu 20.12.2009]. Saatavissa: http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

Wilde, N. & Huitt, R. 1992. *Maintenance support for object-oriented programs.* IEEE Transactions on Software Engineering. Vol 18:12. USA: IEEE Computer Society. ISSN 0098-5589. Saatavilla: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1263033.

Willmor, D. & Embury, S.M. 2005. *A safe regression test selection technique for database-driven applications.* Teoksessa: *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05. Budapest, Hungary. 26.-29.9.2005.* USA: IEEE Computer Society. DOI: 10.1109/ICSM.2005.15.

Wirfs-Brock, R.J. 2009. *Design for Test.* IEEE Software. Vol 26:5. USA: IEEE Computer Society. DOI: 10.1109/MS.2009.125. ISSN 0740-7459.

Wissink, T. & Amaro, C. 2006. *Successful Test Automation for Software Maintenance.* Teoksessa: *Proceedings. ICSM 2006. 22nd IEEE International Conference on Software Maintenance. Philadeplhia, PA, USA. 24.-27.9.2006.* USA: IEEE Computer Society. DOI: 10.1109/ICSM.2006.63.

Wong, W.E. & Horgan, J.R. & London, S. & Agrawal, H. 1997. A study of effective regression testing in practice. Teoksessa: *PROCEEDINGS. The Eighth International Symposium On Software Reliability Engineering. Albuquerque, NM, USA. 2.-5.11.1997.* USA: IEEE Computer Society. DOI: 10.1109/ISSRE.1997.630875.

W3C (World Wide Web Consortium). 2004a. *Web Services Architecture, W3C Working Group Note.* [Verkkodokumentti]. [Viitattu 13.12.2009]. Saatavilla: http://www.w3.org/TR/ws-arch/.

W3C (World Wide Web Consortium). 2004b. *Web Services Glossary, W3C Working Group Note.* [Verkkodokumentti]. [Viitattu 10.4.2010]. Saatavilla: http://www.w3.org/TR/ws-gloss/.

WS-I. 2010. *Web Services Interoperability Organization.* [Verkkosivusto]. [Viitattu 11.4.2010]. Saatavilla: http://www.ws-i.org/default.aspx

Wu, Y. & Chen, M-H. & Kao, H.M. 1999. *Regression testing on object-oriented programs.* Teoksessa: *Proceedings of the 10th International Symposium on Software Reliability Engineering,*

1999. Boca Raton, FL, USA. 1.-4.11.1999. USA: IEEE Computer Society. DOI: 10.1109/ISSRE.1999.809332.

Wu, Y. & Offutt, J. 2002. *Modeling and Testing Web-based Applications*. [Verkkodokumentti]. [Viitattu 19.3.2010]. Saatavilla: http://cs.gmu.edu/~offutt/rsrch/papers/webmodeltr.pdf

Yeh, T. & Chang, T.-H. & Miller, R.C. 2009. *Sikuli: using GUI screenshots for search and automation*. Teoksessa: *Proceedings of the 22nd annual ACM symposium on User interface software and technology. Victoria, Kanada. 4.-7.10.2009.* USA: ACM. DOI: 10.1145/1622176.1622213.

Yuan, H. & Xie, T. 2006. *Substra: a framework for automatic generation of integration tests*. Teoksessa: *Proceedings of the 2006 international workshop on Automation of software test. Shanghai, Kiina. 23.5.2006.* New York, NY, USA: ACM. DOI: 10.1145/1138929.1138942.

Zheng, J. & Williams, L. & Robinson, B. 2007a. *Pallino: automation to support regression test selection for cots-based applications*. Teoksessa: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. Atlanta, GA, USA. 5.-9.11.2007.* New York, NY, USA: ACM. DOI: 10.1145/1321631.1321665.

Zheng, J. & Williams, L. & Robinson, B. & Smiley, K. 2007b. *Regression Test Selection for Black-box Dynamic Link Library Components*. Teoksessa: *Proceedings. Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques. Minneapolis, MN, USA. 20.-26.5.2007.* USA: IEEE Computer Society / ACM. DOI: 10.1109/IWICSS.2007.8.